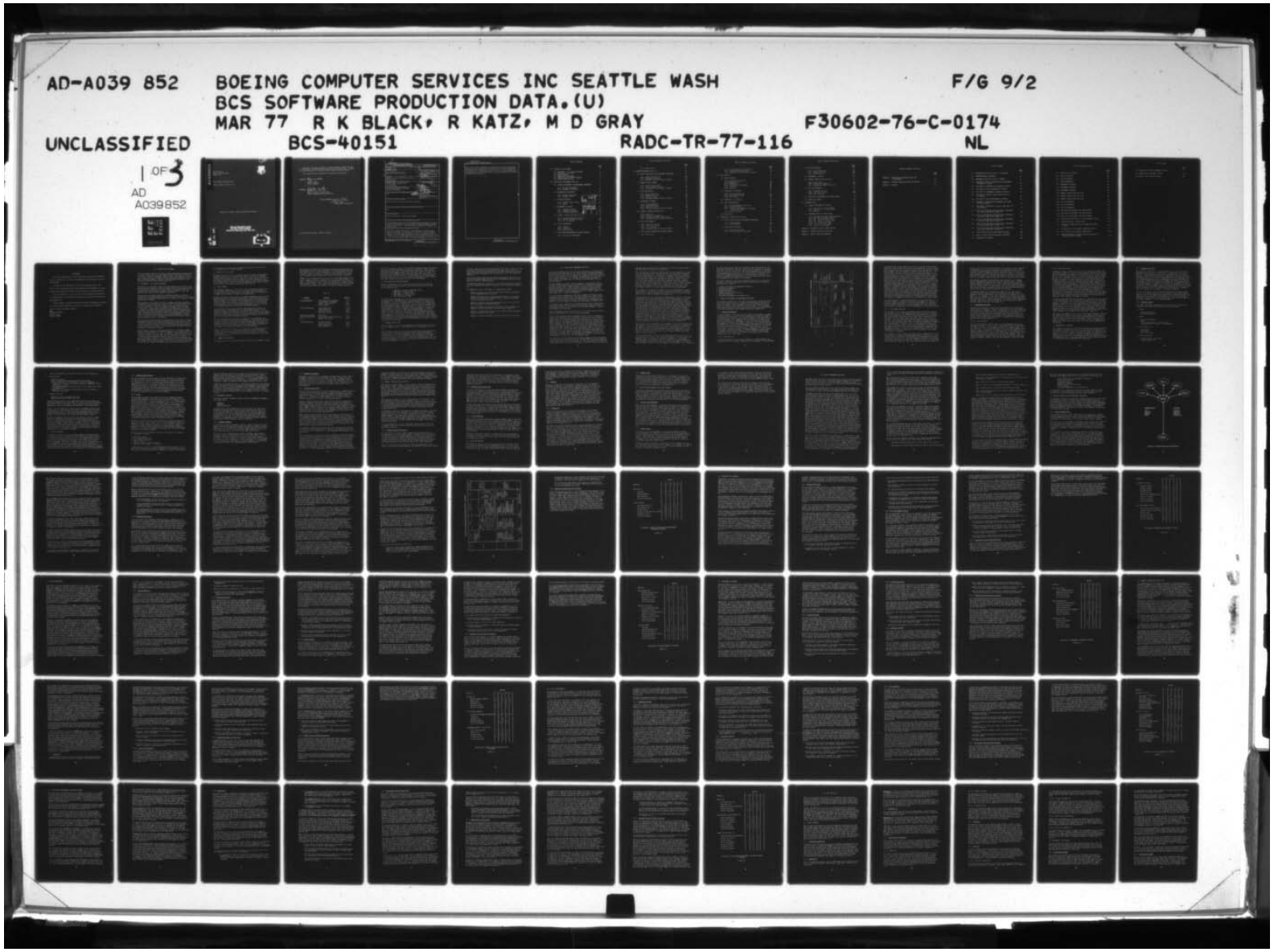
AD-A039 852 BOEING COMPUTER SERVICES INC SEATTLE WASH F/G 9/2
BCS SOFTWARE PRODUCTION DATA.(U)
MAR 77 R K BLACK, R KATZ, M D GRAY F30602-76-C-0174
UNCLASSIFIED BCS-40151 RADC-TR-77-116 NL

1 OF 3
AD
A039852



(12)

RADC-TR-77-116
Final Technical Report
March 1977

BCS SOFTWARE PRODUCTION DATA

Boeing Computer Services, Inc.

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Roger W. Weber

ROGER W. WEBER
Project Engineer

APPROVED: Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-77-116	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): BCS SOFTWARE PRODUCTION DATA,	5. TYPE OF REPORT & PERIOD COVERED: Final Technical ~~Report~~ rept. Feb 76 - Feb 77,
	6. PERFORMING ORG. REPORT NUMBER: BCS - 40151
7. AUTHOR(s): Rachael K. E. Black, Richard P. Curnow, Robert Katz, Malcolm D. Gray	8. CONTRACT OR GRANT NUMBER(s): F30602-76-C-0174 NEW
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Boeing Computer Services, Inc. P O Box 24346 Seattle WA 98124	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F 55810260
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIM) Griffiss AFB NY 13441	12. REPORT DATE: March 1977
	13. NUMBER OF PAGES: 204 192 P.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Roger W. Weber (ISIM)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Modern Programming Practices, Project Organization and Management Procedures, Documentation Standards, Design Methodology, Programming Standards, Support Libraries and Facilities, Testing Methodology, Configuration Management and Change Control.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report contains the results of work accomplished by Boeing Computer Services for AF RADC. The purpose of this study was to assess the impact of modern software development techniques on the cost of developing computer software. The five in-house projects selected for study varied in size type of application, and computing environment. The collection of practices found to have the most beneficial impact on software development are, in order of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

390340 EAH

their impact: Project Organization and Management Procedures, Testing Methodology, Configuration Management and Change Control, and Design Methodology. Existing military standards and specifications are sufficiently comprehensive to encourage the use of beneficial practices; however, certain standards and specifications may require modification to make their applicability to software procurements more pertinent.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF TABLES

EVALUATION

This report describes the software development technology and management practices employed on four specific developments by Boeing Computer Services, Incorporated.

The intent of the RADC program to which this document relates, TPO V/3.4, is to describe and assess software production and management tools and methods which significantly impact the timely delivery of reliable software.

The study contract is one of a series of six with different firms having the similar purpose of describing a broad range of techniques which have been found beneficial.

RADC is engaged in promoting utilization of Modern Programming Technology, also called Software Engineering, especially in large complex Command and Control software development efforts.

Roger W. Weber

ROGER W. WEBER
Project Engineer

1.0 INTRODUCTION AND SUMMARY

This final report contains the results of work accomplished by Boeing Computer Services (BCS) under contract F30602-76-C-0174, "BCS Software Production Data" for Rome Air Development Center (RADC). The purpose of this study was to assess the impact of modern software development techniques on the cost of developing computer software. The period of performance for this contract was February 1976 through February 1977.

1.1 BACKGROUND - THE TECHNICAL PROBLEM

The history of the software industry has been marked by cost overruns, late deliveries, poor reliability, and user dissatisfaction. While these problems are not unique to computing, the record seems to indicate that software developers as a group are less successful in meeting quality, cost, and schedule objectives than their hardware counterparts.

The Air Force, as a major consumer for computing software, is actively interested in the problems experienced by the industry. Reports of Air Force expenditures for data processing establish the annual cost of computer software as over $2 billion/year.

Recent advances in the state-of-the-art in computer software development techniques hold great promise for reducing the Air Force expenditure. Some of these techniques are now being applied within the software industry; they are collectively referred to as Modern Programming Practices (MPP) in this report. These practices have been instituted largely as a result of changes in the computing business environment. Modern Programming Practices include project planning and organization techniques, design methodology, coding and testing practices, use of programming support tools and libraries, documentation standards, configuration management and change control techniques, and the procedures and guidelines necessary to employ these practices in a disciplined manner.

The Air Force requires quantitative information about the effects of Modern Programming Practices on software development cost. Those practices found effective in reducing costs can then be standardized into better specifications and guidelines. In addition the Air Force can encourage contractors to employ beneficial practices in developing software, by establishing the appropriate procurement environment.

Boeing Computer Services, a wholly-owned subsidiary of The Boeing Company devoted to the development of computing systems, is one of many contractors building software for the Air Force. In response to the problems which the industry as a whole has experienced, BCS has undertaken a deliberate effort to improve software development methods. The result of this effort is a set of techniques collectively called Systematic Software Development and Maintenance (SSDM). SSDM as a total concept has not yet fully evolved; some of the SSDM techniques have not been fully tested in the project environment at BCS. The set of Modern Programming Practices described and evaluated

in this study have actually been implemented on projects; these practices are incorporated in the SSDM methodology.

1.2 OBJECTIVE OF THIS STUDY

The objective of this study is to assess the cost effects of various Modern Programming Practices on a representative sample of software development projects. The projects selected for study vary in size, type of application, and computing environment. While the number of projects surveyed is small, and the data gathered for analysis was sparse and ill-conditioned, we believe the findings of this study are significant enough that they can be extended to most software developments undertaken by the industry.

1.3 STUDY APPROACH

The fundamental hypothesis of this study is that software development (definition, design, construction) rules, if rigorously defined and applied and supported by modern techniques and tools, make possible the production of higher quality software at lower than customary cost.

We began our study by defining the set of Modern Programming Practices being employed within BCS, and contrasting these with traditional software development approaches. For each Modern Programming Practice, we determined what tangible evidence would indicate formal implementation of the practice by a software project. We then postulated what qualitative effects these MPP could be expected to have on the software development cycle. This resulted in a rationale predicting the effects of the various practices.

From the prediction of effects, specific hypotheses were formulated for testing and study. These hypotheses, the defined MPP indicators, and the parameters necessary to determine project costs formed the basis for a questionnaire which was used to collect the data for this study. The questionnaire was used in a series of interviews with the Program Managers and key individuals on five BCS software development projects.

Once the data-gathering interviews were completed, the development costs for the five projects were predicted. We based our predictions on estimating techniques currently in use within BCS which presume that traditional approaches to software development are used. We then compared predicted costs for the projects versus the actual costs they experienced, to determine if the projects enjoyed particular benefits from employing MPP.

Since all the projects surveyed did not employ all of the MPP being investigated, the next step was to correlate practices actually implemented on each project with that project's cost benefits. The result of this correlation was a set of MPP determined to have high leverage in reducing software development costs.

1.4 SUMMARY OF CONCLUSIONS

The conclusions of this study are detailed in Section 4.5; a summary follows.

We found that there were four categories of Modern Programming Practices which appeared to be of significant benefit in reducing software development costs. Table 1-1 lists the categories in order by potential benefit, along with some of the specific practices in each category. (Also shown are the document section numbers where these practices are discussed.)

Common to all of these practices is the role of the Program Manager, the person charged with overall responsibility for developing a project's software. It appears that the focus given to a software effort by bringing it under a single manager is key to achieving the benefits of Modern Programming Practices. In fact, there seems to be good evidence that many of the practices we found beneficial would not be possible without that focus.

TABLE 1-1

Category	Practice	Section
Program Management	Task Planning and Assignment	3.1.1
	Status/Progress Assessment	3.5
	Product Identification	3.3.1
	Formal Reviews	3.1.2
Testing Methodology	Testing Objectives	3.6.3
	Testing Formality	3.6.3
	Acceptance Testing	3.6.1
	Functional Testing	3.6.2
Configuration Management and Change Control	Baselining	3.7.1
	Problem Reporting and Resolution	3.7.2
	Change Control Board	3.7.3
Design Methodology	Top Down Approach	3.3.3
	Design Verification	3.3.2
	Design Completeness	3.3.1
	System Planning	3.1.2

Of the five projects surveyed, the three which implemented Modern Programming Practices in a disciplined manner experienced significantly lower actual costs than one would expect had they used only traditional practices. These projects incurred somewhat greater than traditional costs in the earlier portions of the development (Definition and Design), and significantly lower than traditional costs in the later (testing) portions of the development. The lower costs incurred in the later phases of development clearly overbalanced the increased costs associated with the earlier phases.

1.5 IMPLICATIONS FOR FURTHER RESEARCH

Specific recommendations for further research are detailed in Section 6 of this report. Briefly, we believe that additional work should be done in the following areas:

- o Management Guidance on MPP Application
- o MPP Impact on Product Quality
- o MPP Impact on Schedule Risk
- o MPP Impact on Lifecycle Cost
- o Cost Estimating for MPP

The significance of management's role in realizing maximum benefits from the application of MPP implies that formalization of specific guidance for managers of software development should be pursued. The impact of MPP on software product quality, particularly as it is reflected in demonstrated responsiveness to user requirements, should be directly assessed. The effects of employing MPP to reduce schedule risk, especially in the critical testing phases of software development, should be examined. Although this study considers the effects of MPP on software development costs, there remains the larger question of the effects of MPP on software lifecycle costs (i.e. costs of operation, maintenance, and enhancement) which should be the subject of further study. Also, this study shows that projects employing MPP spend less resources, and spend them differently, than commonly-used development estimating guidelines forecast; additional study is required to precisely quantify the effects of MPP and appropriately adjust the industry's estimating techniques.

1.6 DOCUMENT ORGANIZATION

Section 2 documents traditional programming practices employed by BCS software development projects. These practices are, in many cases, still in use.

Recent changes in procurement practices and BCS' operating procedures have resulted in new techniques being applied on BCS software developments. These changes and the Modern Programming Practices employed by BCS are described in Section 3. Not all BCS projects use the complete set of MPP described; most use a combination of traditional and modern practices. This section also documents, for the five BCS projects surveyed for this study, which of the MPP were used by each.

A detailed description of the study approach is provided in Section 4, along with descriptions of the data gathered for analysis, and the results of that analysis. Conclusions concerning specific MPP determined to have major impact on software development costs are detailed in the final portion of Section 4.

Section 5 provides a comparison of the MPP implemented at BCS with the set of practices documented in the IBM Structured Programming Series for A.F. RADC (A.F. Contract F30602-74-C-0186).

Recommendations based on the results of this study are detailed in Section 6. These recommendations suggest areas where further study may be appropriate.

The appendices to this final report are organized as follows:

Appendix A contains key military specifications which are discussed in the recommendations of Section 6.

Appendix B contains the estimating guidelines used in this study to forecast the traditional or expected costs of software development for the five projects surveyed.

Appendix C contains a specimen questionnaire used in this study to gather data about the surveyed projects.

Appendix D contains the predicted traditional costs for each of the projects surveyed, according to the guidelines in Appendix B.

Appendix E contains descriptions of the five BCS software development projects surveyed in this study.

Appendix F contains a glossary of the key terms used in this report.

2.0 TRADITIONAL PROGRAMMING PRACTICES

Traditional Programming Practices are those techniques and procedures which have been (and in some cases are still being) used in software development at BCS. These practices form the foundation for Modern Programming Practices. To understand the implications of Modern Programming Practices (see Section 3.0), we must, therefore, first describe the traditional environment in which software development occurs in BCS and the methods applied within that environment.

The Boeing Company, and most aerospace companies, is organized in matrix form with a functional and project structure. The functional organization generally groups together common business functions (marketing, engineering, manufacturing, etc.), while projects are generally established for handling development of new products and investigating departures from traditional business. Project organizations within The Boeing Company take three forms; staff, intermix and aggregate.

In a staff project organization, the project manager is provided a staff to exercise control through activities such as scheduling, task and funds supervision, and to carry out any functions unique to the project, like testing or site activation. Functional organizations are delegated primary tasks by the staff projects in engineering, procurement and manufacturing.

An intermix project organization is established when some of the personnel who perform primary tasks are removed from functional organizations and are assigned to report in a direct line to the project manager, in parallel with project staff functions.

Under an aggregate project organization, all activities required to accomplish a project report directly to the project manager.

Prior to the formation of Boeing Computer Services, Inc. as a subsidiary, its services were provided by a functional organization within The Boeing Company. As a functional organization, and during the first years after BCS's formation, its primary role was that of a supplier of software development resources and computing services to other functional organizations, and to projects. The functional organization's (and, subsequently, BCS's) primary responsibility was to provide to Boeing functional organizations and projects computing services in a cost-effective fashion, and to provide, augment and retain appropriate software development skills at a level consistent with their current and future needs. The software development resources were (and are) applied in two ways; "on loan" to the using organization, or as a "team" under the administrative management of BCS (or its functional predecessor). With the team approach, technical, cost and schedule management is provided by the using organization.

BCS continues to provide services to The Boeing Company on this basis; Boeing allocates an annual fixed dollar budget, for which BCS provides the computing resources and skills Boeing requires for its functional and project organizations. Boeing determines priorities amongst its various functions and projects.

and specifies the types of skills required to fulfill their needs, which BCS then supplies or acquires as appropriate.

BCS' relationship with Boeing clients has been that of responding to a need the client defines. Prior to approaching BCS, the customer has recognized a need for some capability: for example, a computer-controlled navigation function on board an aircraft. His need, or problem, is expressed in terms of the functional capability which must be available in order to fulfill his total project objectives. Furthermore, knowing his project schedule, the client specifies when the identified capability must be provided, in order to ensure timely integration of this function with the other (hardware) pieces of the system for which he is responsible. In effect, the client has completed concept definition and systems analysis, and comes to BCS with a specification of the requirements to be satisfied by software.

BCS' obligation, as provider of computing services and developer of computer software, is to produce the computer code which implements the required capability on a specified hardware configuration, and to provide that capability for the client on the date he has specified.

Two key characteristics of software development result from this traditional relationship and influence the methods by which BCS discharges its obligations. First, since the client focuses on a functional capability which he requires, the software which implements that capability appears "buried" within his total system, and is not recognized as an identifiable, separate end-item. Echoing this point of view, BCS software developers have traditionally concentrated their planning on the activities (designing, coding, and testing) involved in producing software, rather than on the system which uses software as a constituent part. Second, the development of software is, in the eyes of the client, only a part of the total system development cycle for which he is responsible. Faced with a problem which he requires a computing system to solve, the client is most interested in having the solution available to him in time to address the problem properly. This emphasis on timeliness of solution causes the software developer to concentrate his efforts toward meeting schedule; both the client and the software developer consider product quality, completeness, and ease of implementation subsidiary concerns to the timely delivery of a required capability.

In this traditional relationship, cost management is a client responsibility. BCS supplies its resources to the Boeing client in essentially a "cost-plus" environment. Further, the Boeing user of BCS' resources and services has not been free to procure his needed capability from another source. If the BCS "price" exceeds the client's budget, either he re-examines his requirements in order to reduce the level of effort needed, or does without the capability he has specified.

Given this environment, the traditional approach applied by BCS to fulfill its obligations is to partition the required capability into basic functional components which are assigned to groups of programmers for development (design, coding, and checkout); then to integrate the resulting components into a package of software which provides the required capability. The software development "team" is therefore composed of "programmers" and "integrators".

The traditional approach results in a software life cycle similar to that displayed in Figure 2-1. This model tends to stress the parallelism of design, code, and testing activities, while delaying production of documentation until just prior to delivery. The tasks accomplished during the Operation and Maintenance portion of the life cycle include, in addition to the completion of supporting documentation, the operation of the capability, necessary housekeeping functions such as data set manipulation, and repairs and enhancements to the capability as necessary.

The specific practices traditionally employed within this life cycle are discussed in the subsections which follow. The practices are categorized and presented in terms of:

- o Software Development and Management Procedures
- o Documentation Standards
- o Design Methodology
- o Programming Standards
- o Support Libraries and Facilities
- o Testing Methodology
- o Configuration Management and Change Control

2.1 SOFTWARE DEVELOPMENT AND MANAGEMENT PROCEDURES

As described in Section 2.0, the traditional approach to software development can be characterized by 1) a concentration on computer code as the primary end product, and 2) an emphasis on the delivery date of the end product as the overriding commitment of the developer. In the paragraphs below, these characteristics are illustrated in traditional practices involving software development and management procedures, respectively.

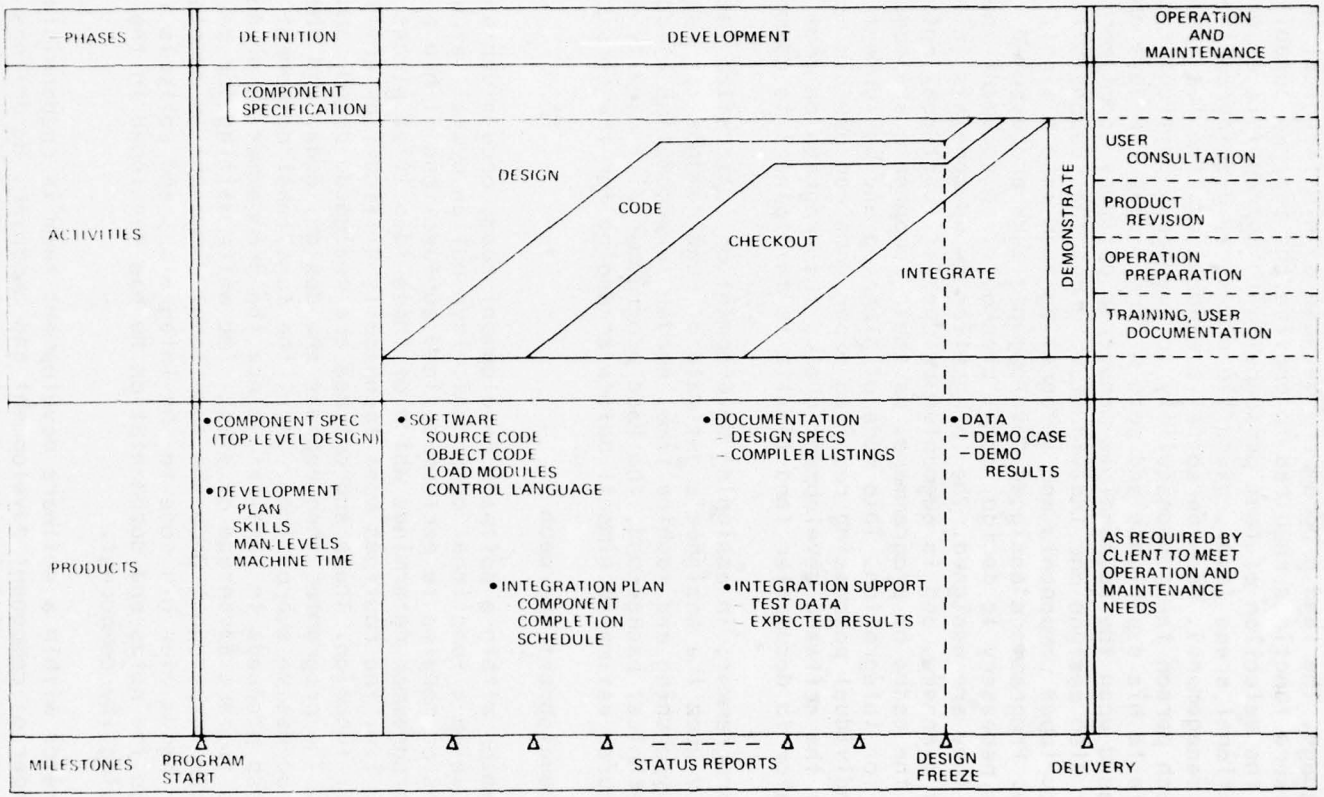
2.1.1 Software Development

The Boeing Company client identifies the problem and the capability needed to solve the problem in the traditional approach to software development. The delegation of responsibility for the development of software which will provide the capability is dependent upon the type of client organization and the preference of its management. For staff projects and most Boeing functional organizations, development responsibility will be delegated to BCS management. Intermix and aggregate-type project organizations and some functional organizations retain the management responsibility for development and acquire selected personnel or skill types "on loan" from BCS. In either case, a team of two or more people, having the skills desired by the project's responsible manager, is formed. Management practices applied are dependent upon the prior experience and performance of the manager responsible for development. For projects, preference is usually superseded by contractual requirements.

2.1.1.1 Team Structure

A software development team is usually headed by a "lead programmer" who reports directly to the manager responsible for development. A lead programmer

FIGURE 2-1. PROGRAMMING PRACTICES LIFECYCLE — TRADITIONAL



is selected, based on his technical expertise in the particular software application (e.g., command and control, digital modeling, financial reporting) and his familiarity with the specified hardware configuration and computer language. The lead programmer, because of his technical skill in the basic software functions required to provide the desired capability, can influence the selection of team personnel with appropriate experience in those functional areas (e.g., executive software, display/control software, data base management). Once personnel are acquired, the lead programmer assigns each person the responsibility to develop a particular component appropriate to his experience and sets a target date for development completion based upon the responsible manager's delivery commitment. The lead programmer also assigns one individual or group the responsibility for receiving developed components and integrating them into a final, packaged capability. Programmers assigned development tasks are assumed to possess the skills necessary to design, code, checkout, and document the software component they are assigned. The integration group consists initially of a few test planners, and is augmented by special functional software expertise from the cadre of programmers, as their components are completed and submitted for integration. This mode of planning and assignment presumes that an individual possessing required experience can perform many tasks throughout the software development cycle; the progression from developer to integrator to documenter (and finally to maintainer) is common practice.

The lead programmer, in assigning development or integration tasks, relies principally upon the assignee's estimate of requirements for resources, such as keypunching and machine time, needed to accomplish a task. Using his own technical background, the lead programmer will usually validate these resource estimates himself before arranging for the services required.

2.1.1.2 Development Approach

The programmer within a software development team, once given an assignment to develop a functional component, lays out an overall structure of subroutines or modules to perform defined subfunctions within a component. Then the programmer determines which of these identified pieces are the "kernels", i.e. the routines most technically difficult or critical to the overall function. These subroutines are designed, coded, and checked out first. The programmer then repeats the design, code, and checkout process for successive subroutines, until the functional component is complete. This process proceeds in parallel, since the programmer is planning his work so as to take advantage of slack time while waiting for keypunching and computer turnaround. Once the functional component is complete, the programmer hands over his code to the integrator, and collects for formal publication the notes and documentation he has developed in the process of developing the component.

The integrator within a software development team is responsible for defining the order of component development and handover. He defines this order in such a way as to minimize the development of "throw-away" test drivers which may be required to tie functional components together. The integrator then devises test data cases which will exercise all of the various options of the functional capability; these test cases may consist of "synthetic"

data or (pieces of) "live" data provided by the customer. As each component is handed over to him by the programmer, the integrator executes it with previously-received components, using the test data cases, in order to determine if the set of components perform together as expected. If any interface incompatibilities are discovered during test execution, the integrator modifies the components as required. The integrator continues this process, until all functional components have been received, exercised, and corrected.

The software development team determines that the capability is complete when all components are present and function smoothly together. A common working definition of completeness is that, in exercising the capability with the complete repertoire of test cases, the software "does not unexpectedly halt, loop, or exit". An additional criterion for determining completeness is that the results obtained from the test runs match hand-calculated results derived from the input values used as test data.

The completed capability is then made available to the client. Usually delivery involves a demonstration of the capability, and hand over of a package of instructions, control cards, etc. required to operate the software. It is not uncommon for one or more members of the software development team to be "on loan" to the user for an interim period as usage consultants. The final task for the software developers is to produce a formal Maintenance Document containing all the information they feel is pertinent to explain, describe, operate, and modify the capability.

2.1.2 Management Procedures

Reviewing and reporting of project activities concentrates on performance against schedule. Progress, in terms of completeness of the total capability, is based upon the judgments of the lead programmer and the technical personnel to whom specific tasks have been assigned. Usually, the schedule is structured in terms of specific development tasks (the production of independent software components), with as much time as possible allotted for the integration process (the consolidation of independently-developed components into a working capability).

2.1.2.1 Status Reviews

To review the status of specific tasks, the lead programmer obtains the assignee's assessment of his progress to-date against the total development or integration task. If the assignee feels that his progress to-date is such that he will meet his task completion date, the lead programmer assumes that performance to schedule is satisfactory, and that the supporting resources planned are sufficient for task completion. If the assignee indicates that he anticipates difficulties in meeting his task completion date, the lead programmer requests permission of the responsible manager either to allow the assignee to spend overtime on the task, or to acquire additional resources (manpower, computer time, keypunch) necessary to meet the schedule.

2.1.2.2 Milestone Reviews

In addition to these informal status reviews, a software development team traditionally conducts milestone reviews to apprise the client (both system users and management) of progress in a more formal fashion. There are two points in the development cycle when milestone reviews are usually conducted. The first is a design review, held after the principal functional components of the capability have been identified, and the development and integration schedule for these components has been planned. The second is an acceptance review, held after the capability has been integrated into a functioning system. The objective of the design review (commonly called a Preliminary Design Review, PDR) is to apprise the client of the nature of the functional components the developers determine are necessary to perform the required capability, and to provide the client assurance that all of the functional components will be developed and integrated on a schedule which will provide his capability on the date he requires. The objective of the acceptance review is to effect the hand over of the completed capability to the client. As described in Section 2.0, this hand over usually involves a demonstration and the delivery of basic materials the client will need to operate the capability himself.

In projects associated with DoD procurements, a second design review (called Critical Design Review, CDR) is required. The objective of this milestone review is to present to the client the details of the design of functional components identified at the Preliminary Design Review. Design details are presented so that the client understands the format of input data required and the type of output which the capability will provide. Since the traditional development life cycle permits staggered development of functional components, the Critical Design Review may be phased so that groups of functional components are reviewed as their designs are completed.

When supplying a capability to a client, the software developers traditionally develop and integrate their product on the computing hardware where the program will be used. In the case of DoD procurements, the required capability may be used on computing hardware different from that used to effect development. Delivery, in this instance, involves transporting the capability to the client's site, as well as installing and demonstrating the package.

2.2 DOCUMENTATION STANDARDS

As discussed in Section 2.0, the traditional approach to software development considers documentation an activity distinct from and subsidiary to the primary objective, delivery of a specified capability. Documents, when produced, are oriented towards "recording the history" of completed development tasks. The criticality of complete and usable documentation for subsequent system operation and maintenance tends to go unrecognized during the development cycle. In the paragraphs below, the pertinence and content of traditional software documentation are described.

2.2.1 Document Pertinence

The traditional objective of software documentation is to fully describe the capability as it exists at the point it is handed over to the client. Figure 2-1 shows documentation activities are delayed until just before delivery. This deliberate scheduling of documentation minimizes the impact of changes occurring in subroutines and components during design and testing. Final documentation tends to be in compendium form, and is developed from the informal documentation which the programmer has created in the process of designing, coding, and checking out a component. This informal material is not available until a component is submitted for integration (or shortly thereafter). Component changes occurring during integration must be reflected in the material the programmer supplies; documentation activities (writing, typing, reviewing, editing, and publishing) are therefore not attempted until the integration of a component is considered complete.

If the client requires material describing how to use the computing capability, it is commonly produced after the capability has been delivered. This User Guide is developed by the member of the software development team "loaned" to the client as an interim consultant, and is oriented toward those specific consultation questions which the client has raised once he has used the capability for its intended purpose.

2.2.2 Document Content

Traditional maintenance documentation is collected into a single volume or volumes, and typically contains the following information:

1. Purpose

   Problem Description
   Development Objectives

2. Method

   Algorithms Implemented
   Scope
   Limitations (accuracy of results, etc.)
   Recommendations (for usage, for enhancement)

3. Input Preparation

   Card Formats
   Deck Setup
   File Descriptions

4. Output Description

   Annotated Report Page Formats
   Display Layouts, etc.

5. Installation/Operating Instructions (including list of diagnostics)

6. Program Details

   Method of Invocation
   Library Subroutines and Operating System Services Used
   Top-Level Organization (relationship of functional components, of components to files and intermediate storage, of components to program input and output)
   Intra-Component Organization (hierarchical list of subroutines, showing dependencies)
   Alphabetical Set of Subroutine Design Specifications, Flowcharts, and Compiler Listings

7. Standard Test Case

   Annotated Listing of Standard Input Deck
   Sample Output Resulting From Input Deck

This Maintenance Document outline has become established since 1962 as a traditional standard in use within BCS. It is similar in content and comprehensiveness to the Air Force Part II Design Specification, which presents standards for maintenance documentation traditional in DoD procurements.

Recently, there has been some recognition that a compendium document such as this may be too comprehensive to be usable, and so portions of the Maintenance Document (e.g. 3, 4, and 5 above) are sometimes broken out into a separate volume called an Installation/Operation Manual. This volume is produced in conjunction with the (abbreviated) Maintenance Document.

The User Guide prepared after delivery of the capability varies in format and content, depending largely upon the needs and sophistication of the client. It usually does not contain all of the information required to exercise the capability and interpret its output, unless the using audience is relatively unsophisticated. Rather, as mentioned earlier, it addresses those specific usage questions which have arisen as a direct result of the client's exposure to the software. Typically, the User Guide includes discussions of how to invoke certain functional options, and how to interpret output reports, error messages, and dumps.

2.3 DESIGN METHODOLOGY

As indicated in Section 2.0, the traditional approach to software design is iterative, and spread throughout the development and integration activity. The Definition phase involves the activity of component specification, i.e. partitioning the specified capability's requirements and allocating them to functional components of software. This is usually performed by the lead programmer. The details of component design are determined initially by the programmer assigned the development task, and may be changed by the integrator as required to make the component execute properly with other pieces of the capability. Specific traditional practices employed in component specification and design activities are discussed in the following paragraphs.

2.3.1 Component Specification

The lead programmer usually possesses technical knowledge of the application for which the client requires a computing capability. Using his knowledge of the application, and the requirements and objectives which the client has communicated to him, the lead programmer decides what major functional components are needed in the completed package. This process tends to be a "top-down" analysis of requirements, resulting in a high-level, skeletal design solution. Once the functional components have been identified, the lead programmer coordinates with his BCS line manager to acquire programmers who possess skills appropriate to the identified functions, and assigns to them the responsibility of building the components.

2.3.2 Design

The programmer assigned to development of a specific component proceeds from the component specification description of functional objective. He subdivides the component function into pieces which will ultimately be subroutines or modules of the completed component. This initial "fleshing-out" of the component design is also a "top-down" analysis, and results in a definition of subroutine functions and relationships. The programmer then determines which pieces are the most technically critical or difficult. Based upon his familiarity with the computing environment in which the capability will be developed (e.g. compiler, operating system utilities, hardware), he proceeds to design, code, and checkout these pieces or "kernels". When the kernels are working to his satisfaction, the programmer proceeds in a "bottom-up" fashion to build from these foundation elements until the complete capability exists. The design of intermediate pieces is shaped by the completed kernels; thus the programmer is using the coding process to reconcile design details within the component structure.

Unless the client specifies the required formats for data input and output, the specification of these formats is part of the programmer's design responsibility. Where these formats involve files to be used in passing data from one functional component to another, the lead programmer typically convenes an informal Interface Working Group involving the programmers responsible for the affected components. This group then resolves the communication protocols between components.

For each subroutine the programmer identifies, the details of its design are developed in terms of:

- Subroutine Flowchart
- Entry Sequence Definition
- Local Storage Organization
- Subsidiary Calls
- Common or Global Storage Definition
- Sizing and Timing Budget (if necessary)

In order to assure that the resulting components perform together, an integration team is established. Completed components are exercised as a package, using test data developed by the integrators. in order to perform their

task, the integrators require the programmer to provide definitions of his component input and output interfaces when he effects hand over. These definitions can then be compared with the definitions for interfacing components already received by the integrators. If interface and communication problems are discovered during integration, it is the responsibility of the integrators to make the design and code changes necessary to correct them.

A benefit of the integrator's role as traditionally constituted is the design visibility available to the client as functional components are completed and handed over. The definitions of component input and output formats supplied at hand over can be made available to the client so that he gains advance visibility of how the complete capability will appear externally; the client can therefore begin to plan how he will use the capability he will ultimately receive.

2.4 PROGRAMMING STANDARDS

Traditional software development projects establish programming standards in three basic areas:

Language
Commentary
Linkage Conventions

The objectives of these standards are 1) the effective use of available hardware and software resources and 2) the consistency, and hence quality, of the delivered capability. These standards result from the recognition of the superiority of certain coding practices in minimizing compilation and execution errors, and the realization that program modification and enhancement are facts of life that demand pre-planning during the coding process. In the paragraphs which follow, the types of standards traditionally defined in these three basic areas are described.

2.4.1 Language Standards

In traditional software development projects, a specific higher order language is usually prescribed as a standard. The choice is based on language availability, type of application (business: Cobol, PL/I; scientific: Fortran, Algol, APL; real time and simulation: GPSS, SIMSCRIPT), as well as client preference (Jovial for the Air Force; CMS for the Navy).

The use of certain language forms or special, perhaps unique, features of specific language implementations may be restricted. Examples include mixed mode computations in Fortran or "direct" instructions in Jovial. Deviations from these standards are typically permitted only when necessary to achieve required efficiencies -- in execution speed or in storage usage -- or to invoke specialized capabilities of the computer or its operating system which the higher order language compiler cannot support.

2.4.2 Commentary Standards

Programmers are traditionally encouraged to annotate their source code with explanatory comments. The lead programmer may leave the amount and type of commentary up to the discretion of the individual programmer. Specific guidelines, when published, typically encourage the incorporation of a description of function, an explanation of symbols, labels, notation, or naming conventions, and a definition of input and output variables within each component.

2.4.3 Interface Conventions

There are three basic kinds of interfaces associated with a computing system: 1) those used for communication between a computer program and its human users and operators, 2) those used for communication between computer programs, and 3) those used for communication within a program. Typically, a project will have formally-defined standards only for the third type of interface, specifically for calls to software subroutines.

Standard linkage conventions (for subroutine call and entry sequences) are ordinarily defined by the computer vendor or the supplier of operating system/executive-level software. These conventions are established to provide a common design basis for various utility functions: debug aids, linkage editor/loader, library subroutines, etc. Because of their pervasive nature, these conventions tend to become de facto project standards; except in rare instances, there is little need for the programmers to devise alternative ways of effecting intra-program communications.

Communications between programs, on the other hand, are not ordinarily subject to vendor-established conventions. These linkages, usually by means of intermediate storage (disk or tape), must be devised to satisfy the needs of the particular communications involved; the variety of possible needs makes it impractical for the project (or vendor) to attempt standardization of all such interfaces. Instead, the affected programmers will usually convene an informal Interface Working Group to identify the required data communications and devise an appropriate protocol.

For man-machine communications, unless the client has specifically defined the input and output formats required, design of the necessary interfaces is traditionally the responsibility of the programmer whose component implements them.

Of all the types of standards traditionally imposed in software development, interface conventions are the most consistently adhered to. This is a direct consequence of the integrator's role in the team environment; the vendor-defined linkages, the programmer's documented input/output format definitions, and the agreements of the Interface Working Group constitute the rules by which the integrator assures that functional components perform together. The integrator is directed to review components he receives for adherance to these rules, and can refuse to accept a component which does not comply. The integrator therefore assumes a quality assurance responsibility for the software developed. In practice, the integrator seldom rejects

a completed component for violation of interface conventions, both to avoid schedule risk and because the integrator can himself change the component so that it conforms. Generally, at the completion of the integration activity, all functional components (have been modified to) adhere to standard linkage conventions, interface conventions, and defined input/output standards.

2.5 SUPPORT LIBRARIES AND FACILITIES

Traditionally, program support libraries and facilities are provided within most computer centers in response to the demand for services common to all computer users. These include programming aids found in: operating systems, text editors, linkage editors, assemblers, compilers, utility libraries, various language subroutine libraries and data structure manipulation packages. System libraries, in particular, containing the standard computation and manipulation routines, are provided for general use in the traditional environment. Editors, assemblers and compilers also provide widespread service.

The underlying rationale for all of these tools is, "If the piece of software required for a job is already available, there is little excuse for building a new one." This thesis, of course, presumes that the tool is capable of performing the required function, that as a result of prior usage it has been proven capable of performing the function reliably, and that it can be acquired more quickly and at less cost than would be incurred developing a new one.

The programmers themselves have generally instigated development of software so configured that it can subsequently be re-used as a utility program, operating system function, or library subroutine. In contrast, it is the project manager (or his subordinates) who must direct the programmers to actually use the re-usable elements available to them. Projects which stress the use of higher-order languages, and projects with particularly tight schedules are typically the most dependent upon the support libraries and facilities available.

In the paragraphs which follow, traditional tools available in the form of standard subroutine libraries, operating system functions, and utility programs are described.

2.5.1 Standard Subroutine Libraries

With the advent of standard linkage conventions (see Section 2.4.3), it has been possible for the software community to begin collecting libraries of subroutines to perform common computing functions, such as calculating the sine of an angle, reading a punched card, or converting a quantity from one representation to another. Continued use of such libraries by programmers has both refined the subroutines previously available and has encouraged development of new contributions to the library repertoire.

While no formal studies of this practice have been conducted in recent years, it is generally acknowledged that the availability and use of standard

subroutine libraries has contributed significantly to improving software reliability while reducing costs. In fact, it is not uncommon for as much as 50 percent of the instructions in an application program to have originated not with the programmer who wrote it, but from the computer system library. Traditionally, software developers expect virtually any computer except the very smallest will come complete with an extensive library of standard subroutines.

2.5.2 Operating System Functions

A similar tradition exists for software which performs the manipulation of a variety of hardware functions, particularly those associated with data input and output. Few programmers except those employed by the hardware manufacturer himself possess the knowledge of equipment operation needed to design, code, and test the control program which can transmit a data record to or from a particular portion of one track on the surface of a specified disk of a multi-disk storage device. Nor, given today's requirements for data integrity and protection, would the manager of a modern computing facility want the average programmer to try.

Further, these same requirements have provided the impetus for the suppliers of "system software" (i.e., those software elements which "come with" the computer) to provide means of invoking these functions as standard elements of higher-order programming languages. Thus, for all but the most sophisticated of applications, there is no need for the programmer to try his hand at the kind of coding which carries with it the greatest risk to the integrity of the system.

2.5.3 Utility Programs

Beyond libraries and operating system functions, another type of tool which has become traditional is the utility program. The most commonly used utilities are those which perform various operations on a data file: compressing the data (usually to collect fragments of unused space and make them available for re-use), adding data to one file from another, deleting unneeded portions of the data in a file, reordering the data (i.e., sorting it), copying the contents of a file to another storage medium (for example, making a backup copy of disk-stored data on magnetic tape), or simply printing the information.

There have been sporadic attempts to develop and use utility programs of less-general capability (usually, to create data files that could be used in testing other software), but these efforts have not been so successful as to become traditional.

2.6 TESTING METHODOLOGY

As discussed in Section 2.0, traditional objectives for software testing are 1) that the software does not unexpectedly fail,--i.e., halt, loop, or exit--and 2) that the output values produced match results of hand calculations from input test data. Satisfaction of these test objectives tra-

ditionally rests upon the integrator and the client, respectively. The traditional approach to testing consists of three steps; checkout (performed by the programmer), integration (performed by the integrator), and demonstration (performed by the integrator if required by the client). These three steps in traditional testing methodology are described in the following paragraphs.

2.6.1 Checkout

Checkout is traditionally considered an integral part of the programmer's development assignment. The programmer is presumed to have the requisite skills to test his subroutines and the functional component which they comprise, at least to the level where the component can be "put in context" by the process of integration. The basic test strategy applied is to use higher-level subroutines, wherever possible, to create the conditions for test. This strategy reduces the amount of resources expended in constructing test drivers, usually considered "throw-away" code. The traditional design practice of constructing "kernel" subroutines first impacts the manner in which the programmer applies this strategy. Since higher-level subroutines which invoke the kernels are constructed after the kernels themselves, the programmer must either write special driver software to exercise the kernels, or (more commonly) defer testing of the kernels until enough of the component structure above them has been constructed to make checkout exercises more convenient (and cost effective).

2.6.2 Integration

A testing strategy similar to that applied in checkout is traditionally applied to the planning and execution of the integration activity. The integrator schedules component hand over in such a way as to allow previously-received components (which are predecessors in the functional execution sequence) to create the conditions whereby a particular component is exercised. In performing their task, the integrators are permitted to alter functional components as required to achieve overall performance. As mentioned before, their primary objective is to assure that the components, functioning together, do not unexpectedly fail.

The second test objective mentioned earlier deserves some special discussion. While it is traditional practice to verify by hand-calculation the results obtained with a particular set of input test data, it is also traditional for the software development team to accept no responsibility for the validity of the algorithms implemented in their code. The only exceptions to this are if the programmers themselves devised the algorithm, or if they had to substantially modify an algorithm provided by the client in order to implement it in computer code. In effect, this attitude on the part of the software developer implies that his traditional role is that of providing a logical structure of computer instructions which house the client's algorithm(s). In other words, the software developer's job is to provide 1) the code which translates the client's algorithms into computable instructions, and 2) the code which reads data, determines which algorithm to apply, and reports the results of application.

2.6.3 Demonstration

Demonstration of the completed capability for the benefit of the client is usually performed in an informal fashion. The software development team traditionally does not design, prepare, and conduct a formal demonstration of the package unless obligated to do so (usually by contract). Typically, the client provides the demonstration data, and is the party responsible for determining if the results the software produces are realistic and correct (see previous discussion on test objectives).

2.7 CONFIGURATION MANAGEMENT AND CHANGE CONTROL

Configuration management, as evidenced by formal nomenclature and identification conventions, is implemented for software when required by the client (usually to satisfy some contractual obligation). Change control is traditionally imposed by "locking up" the code at the time the capability is considered complete, and earlier when functional components are handed over for integration. Traditional software development practices for configuration management and change control are described in the following paragraphs.

2.7.1 Configuration Management

While configuration management is rigorously practiced by other engineering organizations within The Boeing Company, BCS, as supplier of computing services and resources to Boeing clients, has traditionally not implemented the practice. This is a consequence of the observation that, unlike a manufacturer of airplanes, the software developer creates his product only once, and then modifies it as required to meet changing needs. This attitude is reinforced by the traditional mode of operation, in which the computing capability is developed and used at a single site. There have been occasions when the Boeing client has imposed configuration management disciplines on the software product. This is usually accomplished through the expedient of establishing conventions for the naming and identification of items within a configuration. In this way, items can be conclusively specified as belonging to a particular configuration having a particular capability, and can be distinguished from members of other configurations having different capabilities.

2.7.2 Change Control

Traditionally, the lead programmer of a software development delegates to each programmer the responsibility of maintaining an up-to-date copy of the source code, separate from the version which may be undergoing integration or in operational use. The practice of "locking up" source code is imposed at the discretion of the lead programmer, and may be introduced in response to or in anticipation of problems arising during integration or subsequent Operation and Maintenance.

The point at which "locking up" is considered virtually mandatory is when the integrators have determined that the capability is complete. This ensures that consultants loaned to assist the client in using the capability have a baseline from which to diagnose problems.

It is common in traditional software development efforts for the "locking up" activity to occur at an earlier point as well: when each functional component is handed over from programmer to integrator. This ensures that the integrator will have the latest version of the software available to him to modify as required in the process of making components execute together.

One technique often employed in "locking up" software is to entrust the source code library to an individual who is not himself either a programmer or an integrator; ideally, the individual should not possess the skill to devise and introduce software changes himself. Such an individual, provided with written procedures concerning change authorization, access to controlled code, and introduction of new elements into the controlled library, can function as a vital member of the integration team and as a result enhance product integrity.

3.0 MODERN PROGRAMMING PRACTICES

The objective of this study is to determine how the use of Modern Programming Practices affects the cost of developing computer software. In this section, we will describe the practice, state the tangible evidence which would indicate use of the practice by a project, and identify which of the projects we studied exhibited that evidence.

The use of Modern Programming Practices by BCS and The Boeing Company has been prompted by recent changes in the computing business environment and in the posture of BCS. The following paragraphs summarize these changes.

The computing business environment has changed primarily in the area of contracting provisions, and reflects a growing sophistication on the part of the buyers of computer software. Having experienced cost overruns, late deliveries, poor reliability, and user dissatisfaction (which traditional approaches to software development have failed to alleviate), the customer has altered his basic contracting vehicle, has more rigorously defined the end product he is procuring, and has more precisely specified the visibility he requires during the development process. The traditional relationship between developer and customer has been "cost plus"; in order to reduce his exposure to cost risk, the customer has begun to emphasize the "fixed price" and "incentive fee" type of procurement. Traditionally, a software capability has been procured as "data"; to reduce the possibility of user dissatisfaction with an incomplete package, the customer has begun to designate specific end items to be delivered as products of the software development activity. These end items include not only computer code, which has been the traditional deliverable, but also such products as design documentation, user documentation (for operation, installation, training), test plans, test procedures and test data. While traditional contracting vehicles specify formal review points when customer visibility of the progress of the software development is to be provided, these reviews usually have been conducted in an unstructured fashion, and their objectives have not been clearly understood or achieved. These formal reviews are now being recognized as critical to a successful development; their objectives are more precisely stated, and the review of specific products appropriate to those objectives is now performed. The customer now requires visibility of the plans associated with software development, in order to reduce risk to schedule. He also requires more visibility of the details of the software testing activity, in order to reduce the risk of an unreliable product.

The posture of BCS has changed as a consequence of its formation as a separate corporate entity. The Boeing Company, responding to the growing public awareness of computing services, determined that the cadre of software skills which had been assembled to support Boeing tasks should have commercial potential. BCS was established with the charter to supply computing services to the open market, in addition to support for the company. Our relationship with Boeing includes the traditional one described previously (Section 2); in this, we are software resource managers for Boeing clients. In our new role as profferor of our services to the commercial (and DoD) market, we find our responsibility has changed to that of managing contracted activ-

ities. In addition, on some Boeing projects, software development responsibility for specific applications has been organized as a program and subcontracted to BCS.

BCS is organized into Districts and Divisions, according to geographic service areas and customer sets, respectively. The BCS Districts each contain a data processing center and are composed of functional organizations and staff type program organizations. The Divisions use the aggregate program approach with functional organizations directed toward business management activities (i.e. resource accounting, contract administration, etc.).

These changes have significant implications in the way the BCS developer now performs his function. The BCS Program Manager (instead of a lead programmer) must now manage to cost, as well as to schedule and the satisfaction of the customer's need. This results in an increased emphasis on efficient use of resources, and elimination of activities and products not immediately germane to the project objectives. The software development team is now obligated to provide end items other than code, and to provide them at milestones prior to contract completion (see Section 3.1 discussion on reviews). As a developer contracted to provide a product, BCS is now obligated to accept total responsibility for that product. This responsibility includes accuracy, reliability, completeness, adherance to standards, and in some cases, warranty of the capability provided.

The traditional assignment to develop a functional component was assumed to be partitioned into discrete tasks: designing, coding, test planning, technical writing. Further, the tasks relative to a functional component were usually assigned to one person with generalized skills. Instead of acquiring individuals with generalized software production skills in a functional area for the duration of a project (and employing them progressively as developers, integrators, and documentors), the Program Manager now negotiates for the assignment of individuals for just that period of time he can most productively use them. This implies that the Program Manager must select individuals for task assignments on the basis of more specific skills and proficiency.

Also, the Program Manager must plan these tasks in more detail, to minimize non-productive effort on the part of the individual (e.g. arranging for machine time, keypunch services), and in order to collect the tangible products of the task (and verify their adequacy) before the assignee leaves the project. In addition, in those BCS divisions using the aggregate approach to program organization, the Program Manager shares the responsibility of re-assigning individuals to other programs. The result of the foregoing is that the use of resources on a particular program must be more carefully defined, justified, and negotiated.

BCS has introduced a number of changes to its traditional programming practices in order to accomodate these implications. They include:

- More detailed task planning (for efficient resource utilization).

- More specific internal status reviews (in terms of identified products).

- More pertinent documentation (no longer in compendium form).

- More orderly design (emphasis on top down analysis, to allow tailoring of capability to cost).

- More formal code structures (structured logic forms).

- More deliberate use of programming "standards" (for system integrity).

- More rigorous, objective testing (of both designs and code).

- More careful configuration identification (to support auditing of total set of deliverables).

- More explicit quality assurance (before proceeding into integration).

Central to the Modern Programming Practices now being applied by BCS to software development is a re-structuring of development activities around the milestone reviews. New emphasis on three formal customer reviews (Preliminary Design Review, Critical Design Review, and Physical Configuration Audit/Functional Configuration Audit) has resulted in many activities, which were traditionally performed late in the development project or postponed until after delivery, being moved to earlier phases of the schedule. The functions of Definition (of requirements and end item characteristics) are being consolidated into a set of coordinated activities. The Design effort exhibits a new emphasis on planning (for construction, testing, acceptance, installation, operation, etc.) as an important part of the activity. Further, Definition and Design (with Planning) are becoming discrete, sequential phases with definite objectives and end items. The Construction tasks of coding, debugging, and integration testing are still being performed as parallel activities; however, Construction tasks are being more formally defined, and their products are being explicitly specified and reviewed.

The re-structuring of project activities has resulted in a recognition at BCS of the need to consolidate and formalize our Modern Programming Practices into a new software development methodology. This methodology is called Systematic Software Development and Maintenance (SSDM). SSDM as a total concept has not yet fully evolved; some of its techniques have not yet been fully tested in practice. The approach which SSDM advocates is consistent with the changes which have been previously discussed. The basic activities involved in the production of computing software have not changed, but responsibility for the activities, the order in which these activities are performed, and how they are monitored have changed. These changes constitute the Modern Programming Practices currently employed at BCS.

The objective of this study is to determine how the use of Modern Programming Practices affects the costs of developing computer software. To make this assessment, we examined five current BCS projects. They are described in Appendix E, using alphabetical aliases for anonymity. While none of the five projects employed all of the MPP described in this section, and we encountered variations in practice implementation among projects, we believe the following descriptions are indicative of how these MPP have changed

BCS' traditional way of doing business. In the subsections which follow, BCS' Modern Programming Practices will be presented in seven categories:

Project Organization and Management Procedures
Documentation Standards
Design Methodology
Programming Standards
Support Libraries and Facilities
Test Methodology
Configuration Management and Change Control

In each case, we will describe the practice, state what tangible evidence we looked for to determine whether the practice was being employed, and identify (by aliases) which projects exhibited that evidence.

3.1 PROJECT ORGANIZATION AND MANAGEMENT PROCEDURES

As described in Section 3.0, three basic changes have occurred in BCS' computing business environment: 1) BCS has entered the commercial market as an independent entity; 2) the basic contract vehicle has changed from "cost-plus" to "fixed-price"; and 3) the customer has re-defined and re-emphasized formal milestone reviews. How these changes have affected the project organization and management procedures for software development, and the Modern Programming Practices now being used to effect these changes, are described in the following sections.

3.1.1 Project Organization

As described in Section 2, BCS' traditional role in supporting its parent, The Boeing Company, has been to supply resources--programmers and computing services--that Boeing used in its functional and project activities. With few exceptions, this relationship with our prime customer remains unchanged.

With BCS' entry into the commercial marketplace, and on some recent contracts obtained under Boeing auspices, the software developer has had to assume an unaccustomed role: that of Program Manager. That is, BCS is now doing work under an increasing number of contracts where personnel are given total responsibility for what is accomplished with the resources placed at their disposal.

The environment in which a Program Manager operates is shown pictorially in Figure 3-1. Traditionally, BCS has acted as line (or resource) manager, responding to a Boeing project manager's requests by providing personnel (and other services) with skills, knowledge, and experience--in the production of computing software--appropriate to perform the work required to meet the project's objectives. BCS' job was to develop the needed computer programs and related end items that would provide the capability specified by the project manager; the function of BCS' line managers was to locate, qualify, and assign to the effort suitable people to do the work (and to augment, replenish or reassign this staff as necessary).

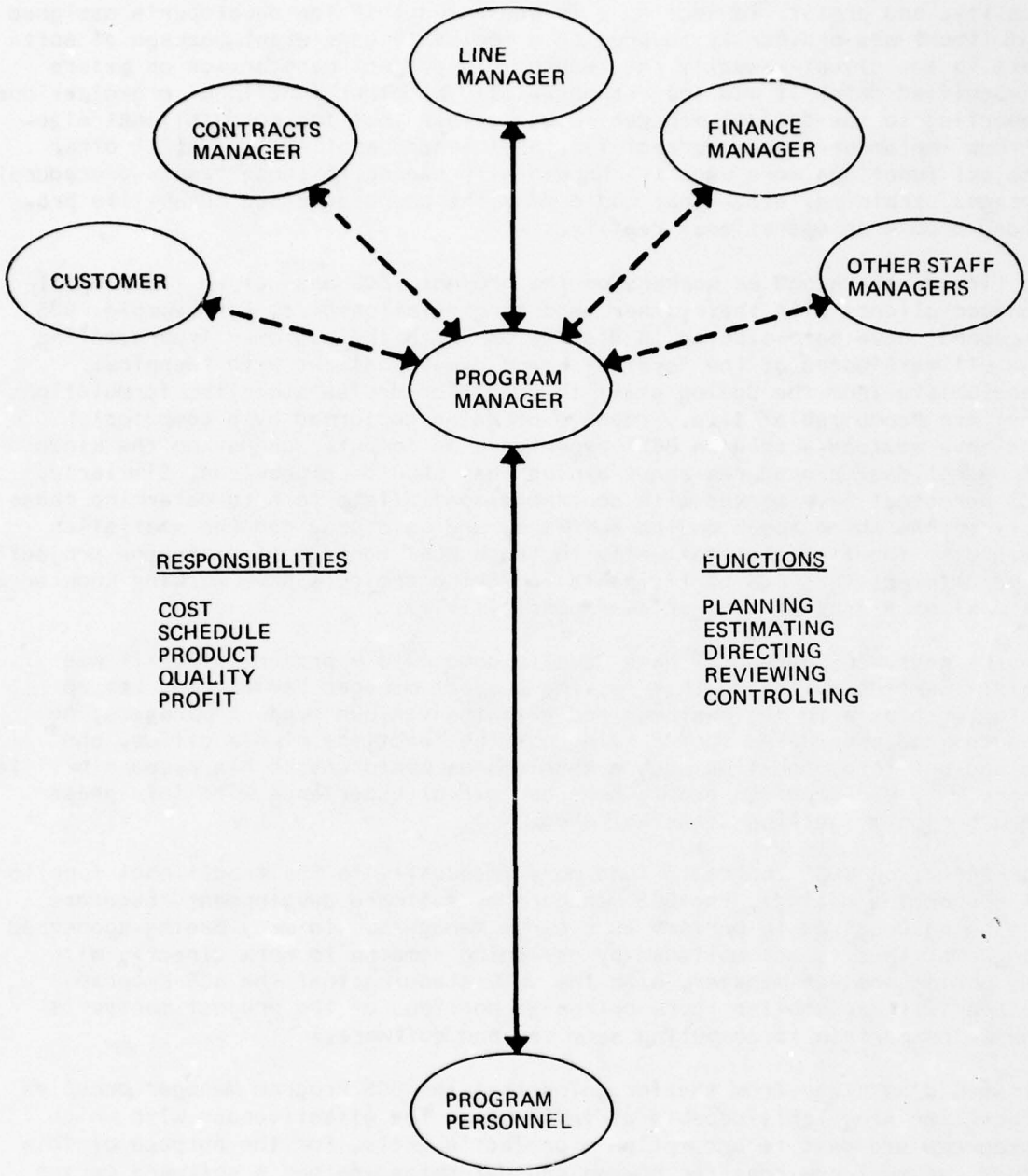


FIGURE 3-1. PROGRAM MANAGER ENVIRONMENT

While BCS has traditionally assisted our client project managers in their functions of planning, estimating, directing, reviewing, and controlling of software activities, the developers have not directly shared the project manager's responsibilities for cost, schedule, product (functionality), quality, and profit. In Section 2 it was noted that the developer's assigned commitment was ordinarily to provide a logically consistent package of software to the client--usually the responsible project manager--on or before a specified date; it was the responsibility of other functional organizations reporting to the project manager to be certain that the computational algorithms implemented were correct for the intended application; still other project functions were usually charged with seeing to those tasks--procedural changes, training, etc.--that would make the computer-aided capability provided become an operational reality.

As line managers and as workers on the project, BCS has helped its project-manager clients with their other reporting relationships. For example, BCS personnel have participated in discussions with the customer (representing the ultimate users of the "system" being developed) and with technical specialists from the Boeing staff to select or devise algorithm formulations that are "computable" (i.e., capable of being performed by a computer), and have advised--based on BCS' experience as computer users--on the kinds of operational procedures and training that should be provided. Similarly, BCS personnel have worked with contracts specialists to help determine those obligations which touch on the software, and have provided the statistics needed by the financial community to track BCS' contributions to the project's expenditures. Thus BCS participants in Boeing projects have working knowledge of most of a project manager's responsibilities.

In the past, BCS personnel have usually come onto a project after it was fairly well established. That is, the project manager had already set up relationships with the customer and with the various support managers, he had devised procedures for carrying out the functions of his office, and he had put into operation such mechanisms as would ensure his responsibilities where met. Very few BCS people have had direct experience with this phase of a project: getting it established.

Currently, on BCS' contracts (and more frequently in the traditional function of supporting Boeing), the BCS managers of software development resources are being required to perform as Program Managers. (In many Boeing-sponsored programs, this is accomplished by assigning someone to work directly with the Boeing project manager, with the understanding that the BCS Program Manager will accomplish those delegated portions of the project manager's job which pertain to computing services and software.)

It should be clear from the foregoing that the BCS Program Manager occupies a position singularly capable of influencing the effectiveness with which resources are used to accomplish a project's goals. For the purpose of this study, we must now consider how we can determine whether a software person in fact occupies such a position.

In order for a Program Manager to discharge his responsibilities, he must have commensurate authority to allocate resources and to supervise activ-

ities. He must have responsibility for meeting cost and schedule commitments, to deliver products and to satisfy quality criteria which he has responsibility to negotiate, and for discharging his obligations in a way that both satisfies the customer's needs and meets company profit objectives. He must be able to secure resources in accordance with his plan for the work, and he must have supervisory responsibility for making assignments, monitoring and directing task execution, and evaluating performance.

Most of these attributes are such that one cannot easily, as an outside observer, determine their existence. For the purposes of our study, then, we chose to structure our inquiry around three basic criteria we felt would indicate that the designated software Program Managers for the subject projects had performed this role rather than serving as traditional resource managers. Specifically, our questions were designed to ascertain whether:

- The Program Manager had both technical and administrative responsibility for (an assigned portion of) the project.

- The Program Manager had (and exercised) authority to discharge his responsibilities, including the authority to make assignments and evaluate performance.

- The Program Manager made formal, written task assignments specifying required products and due dates, establishing resource budgets for accomplishment, and detailing means of production and criteria of acceptability.

3.1.2 Management Procedures

If he is to wield the influence of which the office is capable, it is not sufficient that a person designated as Program Manager merely possess the responsibility and authority required by that position. He must also exercise them. That is, he must do something to ensure that his plans are being followed and that the assignments he has made in accordance with these plans are being executed. Further, he must determine that the results of those plans and assignments are sufficient to meet the goals of the program.

As mentioned in the introduction to this section, the contractual environment in which software is developed has undergone significant recent change. The trend to fixed-price procurements has emphasized the need for careful attention to costs--both in estimating what work must be done and in controlling the use of resources to accomplish that work--if profit objectives are to be met. In addition, customers are requesting, often as stipulations of the contract, that they be given better visibility both of what end items will be produced as a result of the contract and of what activities are planned to produce them.

In the latter case, the contract may call for formal reviews to be held, with customer participation, where the contractor presents information about what has been accomplished to date and what subsequent work is planned to complete his obligations. Such reviews are not new, especially in military procurements, but their application to software is a significant change.

In the past, procurements involving computing capabilities have usually been quite explicit about the required characteristics of the computing hardware--specifying instruction sets, data width, processing speeds, memory capacity. In contrast, the specifications have been vague regarding the software which implemented the required capability--often simply referring to it as "data". With sophistication born of experience, buyers of computing systems (or systems which employ computers) are recognizing that they must define rather precisely the "data" they require to make their computers work and provide the capability needed.

Recall that software developers have traditionally viewed their product simply as a set of machine instructions; the user could then direct the computer to execute those instructions to process his data and produce results appropriate to his needs. The user now sees that he needs not only the instructions, but a variety of other things as well. For example, the contract will now call for delivery of a document describing not only how to direct the computer to execute the software, but also how to prepare the data for processing, and how to interpret the results. The contract may specify that the software be coded in a symbolic language that the customer has people trained to understand (in the event he should later wish to modify the capability). If several users are to employ the capability, the contract may require delivery of materials that will help train them to use it effectively. And there may be a stipulation that, for the contractual obligation to be considered satisfied, once the software has been installed in a designated computer facility, the developer must demonstrate that correct results are produced when data typical of the intended usage are processed.

Formal reviews are one means by which the customer can ensure that the developer is doing his work in a way that will satisfy the contractual obligations. Typically, there will be three such milestones: a Preliminary Design Review (PDR), a Critical Design Review (CDR), and an audit of the physical and functional configurations of end items (PCA/FCA). Usually, the contractor will convene a formal meeting for each milestone review. If (usually because of geographic separation of the developer's and customer's locations) the customer elects not to attend the meeting in person, he will usually ask that the review materials be submitted to him beforehand for examination and comment.

The objective of Preliminary Design Review is for the developer to present evidence, and obtain customer concurrence, that the requirements to be satisfied for a fixed price by the computing capability have been properly identified and, if formal demonstration at delivery is called for, that suitable means of demonstration have been devised. Working with representatives of the using community, the developer will determine what characteristics --functions, performance, interfaces, and environment--the computing capability must possess, and record these findings in a requirements definition document, supported by a prioritization of requirements according to criticality of need and anticipated benefit.

As further evidence that he understands the intended application, the developer may submit the equivalent of a User Guide, describing what the computing capability will look like to the intended user. This material can serve as a vehicle for showing that the software (in conjunction with the hardware)

will in fact satisfy the requirements established for it. In preparing this document the developers will ordinarily have considered also what major software components will be required to implement the capability, and thus will be prepared to discuss how these components can be constructed and integrated. In addition, the developers will have analyzed the costs to satisfy user requirements versus their (criticality and benefit) priorities, and will present information which justifies their recommended design solution based on the contracted price.

Between PDR and the next formal milestone, Critical Design Review, the developer completes his planning for the work he must accomplish to meet his contractual obligations. At CDR, then, he should be able to present a list of the identifiable products that will result from his development efforts. He can state the purpose of each (specifically, how it will contribute to meeting the established requirements) and when it will be completed. Besides the software itself (usually identifiable to the level of individual subroutines), this configuration list will include such items as training materials, test procedures, demonstration data cases, installation aids, and operational procedures.

Some of these listed products will be specified by the contract as deliverable end items; the remainder are constituents of the end items or are otherwise required in producing them. The CDR objective is for the developer to show that there are no critical omissions from this list or, more to the point, that he has planned for everything he will need to complete delivery of the required capability. Once satisfied of the adequacy of the developer's plans, the customer authorizes work to proceed.

The final formal milestone (excepting only delivery, demonstration and acceptance) is the Physical Configuration Audit/Functional Configuration Audit. The PCA portion of this review consists of noting that all items reported at CDR as planned have actually been produced. Usually this simply requires that someone has signed for each item listed, indicating that the item is physically in his possession or under his control. The FCA portion is similar, in that someone acknowledges having inspected or tested each item to determine that it satisfies the function for which it was developed.

The objective of PCA/FCA is for the developer to provide notice that he is, or shortly will be, ready to accomplish delivery and complete his contractual obligations. Further, the customer is given evidence (the signed-off configuration list) of this readiness and need not rely solely on the contractor's judgment for the customer to commit his own resources to accept the product and transition to operational usage.

The effect of imposing formal reviews such as those just described is to substantially restructure the activities of software development. As shown in Section 2, traditionally the development effort has consisted of a repetitive, overlapped cycle of designing, coding, and checking subroutines of a software component, followed by a process of integrating these components into an operational whole. The definition of requirements for the total capability was usually performed by the client prior to the start of development, and refined as the details of the design were resolved. The client performed the final testing of the software, largely to be sure that his

algorithms had been implemented correctly and that they performed as expected.

It was often at this point in traditional projects the client became aware that he needed additional work to be accomplished if he was to make effective use of his new computing capability. Specifically, he needed training materials, operating and maintenance instructions, usage guidance and other similar materials. Since the software developers had completed their work and were available for reassignment, they constituted a ready, in-place means of getting these new tasks done. BCS and its customers now generally recognize that these "additive" tasks are really part of the software development job, and plan for their accomplishment so that they will be completed at the same time the software itself is. As a result, the overall process--when structured by PDR, CDR, and PCA/FCA milestones--comes to look like that shown in Figure 3-2.

In later portions of this section and the next, we will discuss the effects of this restructuring on the manner in which the tasks of software development are performed. In particular, we will see that the Program Manager must change how he determines status and obtains visibility of designs and products on which he bases formal reports to the customer.

The milestone reviews described are typical of military procurements. Similar, not necessarily identical, milestones may be employed in other types of software developments, including those undertaken for the Boeing corporate client.

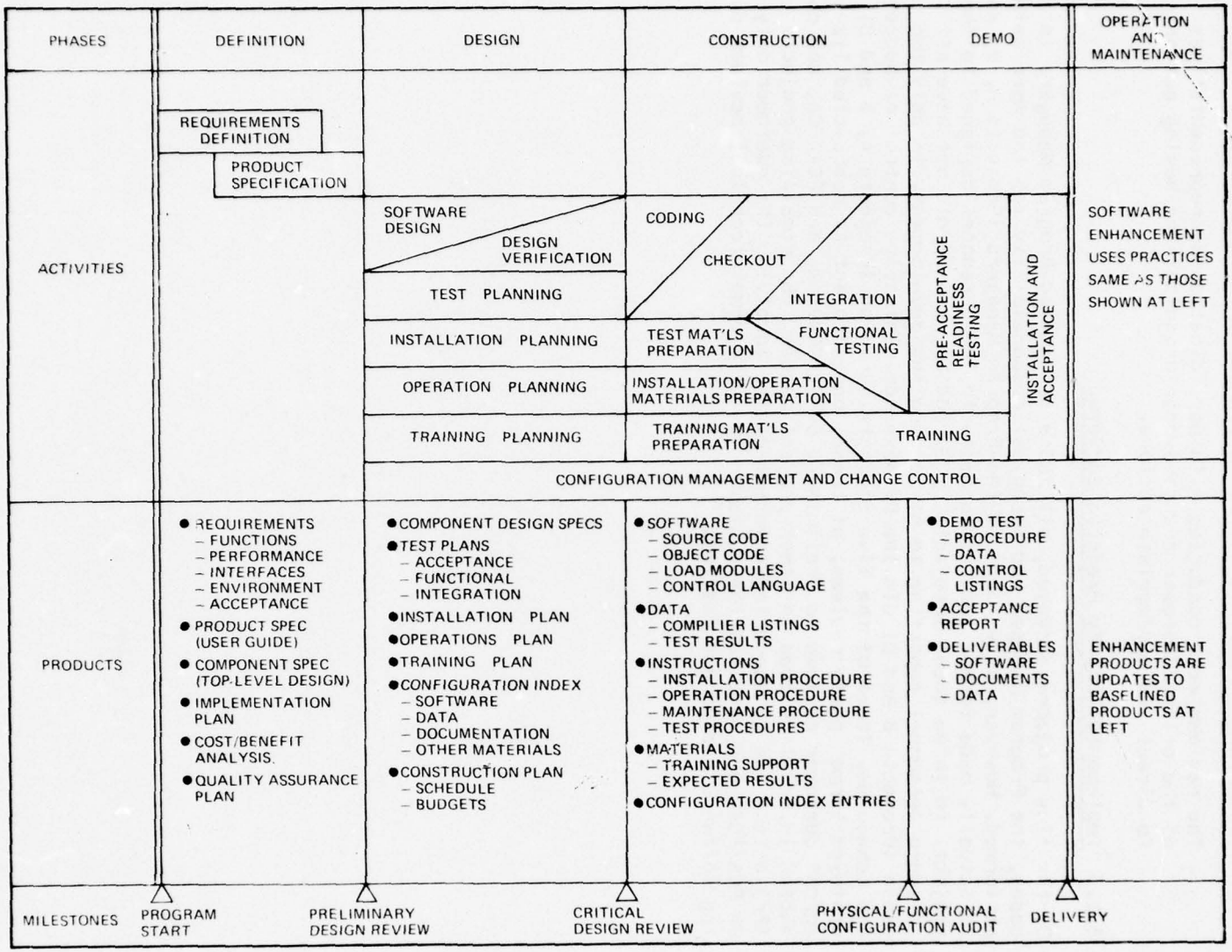
We must acknowledge, however, that such milestones are not a critical prerequisite to restructuring of the process, or to other detail changes in development activities. We believe that the milestones give greater impetus to these other changes, and contribute to making implementation of these changes more effective. We further believe that the fact that a project has employed such milestones is some indication of how well it has used these modern practices and, as a consequence, has provided the Program Manager the information he needs to knowledgably control costs.

We therefore included questions in our survey that would tell us specifically whether milestone reviews were employed and how they were practiced. We assumed that use of milestone reviews would effectively partition the development work into distinct phases, and that the Program Manager would define the expected results of each phase. He would further establish groundrules for the reviews; most specifically, delineating what qualities the review materials should be expected to possess, and what courses of action should be open should these criteria not be met. Finally, we assumed that the formality of the review itself would be a direct function of whether the client/customer participated.

Briefly, then, we looked for evidence that

- The project plan included a schedule of formal reviews, keyed to phases of the development process; specified the products of the phase that were to be presented for review; and established quality criteria for acceptability of these products.

FIGURE 3-2. PROGRAMMING PRACTICES LIFECYCLE – MODERN



- The planned reviews were in fact conducted, and that action was taken to approve, conditionally approve (with assigned responsibility for correction of deficiencies), or disapprove the subject products.

- The reviews were conducted with participation by representatives of the client/customer and developer organizations having authority to direct the appropriate actions.

3.1.3 Implementations By Projects Surveyed

Of the five projects surveyed, all had a designated Program Manager. In all cases, the Program Manager had technical responsibility for the work being performed. However, only three of the Program Managers (Projects A, B, and D) actually made formal task assignments to the personnel assigned to the project. In three cases (Projects C, D, and E), the PM did not have all software personnel reporting to him during the development. In only two cases (Projects B and D) did the PM have administrative control of budgets and schedules. Three of the five projects surveyed (Projects A, B and D) employed formal phase reviews; of these, one (Project A) restricted its formal customer reviews to Preliminary Design Review, and PCA/FCA, and conducted internal reviews in other project phases. The remaining projects (Projects C and E) conducted a formal review involving the customer only at PCA/FCA, and involved the customer in less-formal technical reviews during software design and testing.

	PROJECT				
PRACTICE	A	B	C	D	E
PM Authority					
Technical	X	X	X	X	X
Task Assignments	X	X		X	
Administrative		X	X	X	
Evaluate Performance		X			
Formal Reviews					
End Items Identified	X	X	X	X	X
Review Objectives	X	X		X	
Review Results	X	X		X	
Project and Customer Participate	X	X	X	X	X
Procedure Stated	X	X		X	
Activity Performed	X	X		X	

INDICATORS OF PROJECT ORGANIZATION AND MANAGEMENT PROCEDURES BY PROJECT

FIGURE 3-3

3.2 DOCUMENTATION STANDARDS

Traditionally, software documentation has been produced concurrent with the completion of the software integration activity. Materials from the programmers' workbooks have been assembled into a single, comprehensive Maintenance Document. This approach to documentation has not adequately met the customer's need for timely written material easily comprehendible by the software user, who is typically not a programmer himself. Furthermore, the programmer's workbook, consisting as it does of his own notes, listings, definitions, and flowcharts, has not provided a readily available or useable source of information for dissemination to other project personnel or for use by the Program Manager to obtain visibility of the progress of the development activity.

The programmer's workbook is potentially a useful vehicle for communication among programmers and integrators sharing construction and testing responsibilities. It could also serve as a vehicle for communication with others (including the customer) who have an interest in the planned use, operation, and maintenance of the delivered capability. However, in traditional practice, the programmer's workbook has not been deliberately used for these purposes. It has not been made readily available to individuals other than the programmer himself, nor have standards for its content and format or procedures for updating it been established.

Recent changes in the software procurement environment have caused a re-structuring of documentation to address specific audiences, and additionally require that the Program Manager specifically track the activities associated with document production.

Documents now produced contain essentially the same materials as were available in the traditional Maintenance Document. However, there is now a series of documents, each structured to address specific purposes. The material traditionally developed in support of the programmer's task is surrounded with explanatory text tailored specifically to the needs, interests and background of several audiences. The documents are produced with the realization that the reader of any one of them has a specific functional responsibility related to the use, operation, or maintenance of the computing capability.

The Program Manager is charged with tracking the status and progress of documentation. It is his responsibility to ensure that documents identified as specific end items are produced when the customer requires them. It is also his responsibility to confirm that the documents produced will meet quality standards on which he and his customer have concurred, in order to satisfy the customer's needs. Since the Program Manager is operating in a fixed-price environment, and document production is a significant cost item, he must have visibility of the progress of various documentation activities, to effectively control costs and achieve his profit objectives.

Two Modern Programming Practices are being employed to accommodate these changes: standards and guidance for document pertinence are being applied, and the traditional programmer's workbook is being formalized into a project visibility tool. These practices are described in the following subsections.

3.2.1 Document Pertinence

The new document standards employed in software development projects have as their objective the production of pertinent documentation, i.e. documents tailored to the needs of specific audiences and produced early enough in the development activity to provide necessary visibility (particularly to the customer). This implies that documents must be planned to be produced in parallel with other development tasks (usually in parallel with design activities), rather than planned as the last obligation of the software developer. Document standards are developed and applied which directly address the functional needs of specifically-identified audiences. Furthermore, documents are submitted in draft form to their intended audiences for early review.

It should be noted that the standards described here are not universal; rather, they are project-unique, shaped by contractual obligations, customer needs, and project objectives. In other words, the specific document plans and standards employed may vary from project to project, but they exhibit the general characteristics discussed in the preceding paragraph.

These new document standards may somewhat increase the costs associated with review and packaging. The materials are now more critically reviewed to ensure that they meet the audience's needs; and multiple documents are being packaged for publication instead of the single, omnibus Maintenance Document. However, having a specific definition of form and content of a document, and an understanding of the document's intended use, the programmer should find the preparation task less burdensome and time-consuming (and hence, less costly). Further, key information such as input/output formats, design specifications for components and subroutines, and standard nomenclature conventions is recorded and made easily available to allow the programmer and integrator greater visibility of how the discrete pieces of a software capability relate to each other. This visibility should help each individual to perform his assigned tasks with increased efficiency and less cost.

As has been noted earlier, we did not expect to see identical documentation standards on the projects we studied. However, we did structure our inquiry around criteria which we felt would be indicative of the fact that the standards adopted were pertinent and were employed. Specifically, our questions were designed to ascertain whether:

- Written project procedures established a schedule for documentation and the reviews at which the documentation was due.

- The project justified the need (e.g., contract commitment or internal communications) for each document produced.

- Prior to an activity, the documents which would result were identified.

- Every document produced was used as source material in some subsequent activity.

- Source documents needed for each activity were available at the start of the activity.

- The intended audience participated in the review of each document.

- The review of a document was directed at verifying that 1) the content was appropriate for the intended user and 2) the document was written in the audience's language.

- The feedback from the audience at a document review was analyzed and incorporated, as appropriate, into the final issue of the document.

- The project established the method by which compliance with the above indicators would be determined.

- The project actually employed the above compliance verification method and determined that the indicators were in fact implemented.

3.2.2 Unit Development Folders

The Unit Development Folder (UDF) is an extension of the traditional programmer's workbook, and as such is a part of the new standards for document pertinence discussed in the previous section. The concept of a UDF as a combination repository and visibility mechanism was originated by TRW Systems, Inc. As implemented at BCS, the UDF concept is a standard for documenting plans and progress, as well as for recording key information developed as a product of specific tasks.

As in traditional practice, the programmer responsible for design of a functional component begins by laying out the structure of subroutines needed to complete the component. These subroutines are named, and the programmer then prepares a Unit Development Folder for each, with a cover sheet (or equivalent) which sets a schedule for the completion of design, coding, checkout, and technical documentation. As each of these detailed tasks is completed, the results are placed in the UDF and the date (actually) completed and review signatures (if required) are entered on the UDF cover sheet.

A similar approach is used for test planning. An overview of the (component, integration, or acceptance) test activity is prepared which identifies the various test steps involved and names them. A UDF and cover sheet for each test, scheduling the test design, preparation of test procedures and data, and test execution is prepared. As test designs, procedures, data, and results are produced, they are placed in the UDF and appropriate entries are logged on the cover sheet.

While it has not yet been attempted in practice, it should be noted that the UDF concept can also be applied to producing and controlling plans and materials associated with other activities (that do not directly tie to

software subroutines) such as training, installation, conversion, operation, and maintenance. This extension of the UDF concept seems a likely outgrowth of the Modern Programming Practices currently employed within BCS.

The Unit Development Folder, then, becomes the mechanism whereby a programmer or integrator can easily review his task schedule, report completion of tasks, and record acceptance of results. It establishes naming and referencing conventions for the products of tasks which make these products more accessible to other project personnel. Communication between the individual assigned a particular series of tasks and his superior can be made more efficient (and less costly) since tangible products are available for objective and pertinent review.

The cover sheets of UDFs provide information which the Program Manager can readily extract for project visibility and control. This permits the PM to make an accurate assessment of progress against plan (both schedule and expenditures). The PM can more quickly detect impending performance problems and can effect an earlier (and usually less costly) recovery.

As noted earlier in conjunction with document standards, this practice may be somewhat more costly to implement than the traditional programmer's workbook, primarily because of the emphasis on the cover sheet, which implies more deliberate and detailed review of products for status, adherance to quality criteria, completeness, etc. However, we feel that this cost is more than offset by having this controlling information readily available in a standard form for the people who require it to monitor cost, schedule, and quality; the assignees, their superiors, and the Program Manager.

We structured our project inquiry to ascertain whether the UDF concept was implemented as a project planning and management visibility tool. Specifically, our questions addressed whether:

- Written project procedures established a UDF system and stipulated minimum form and content requirements to be satisfied by each UDF.

- The above procedures specified the method for UDF maintenance which was directed at promoting the UDF as a useful tool in the development of other end items or in the performance of other necessary tasks.

- The project established the method by which compliance with the above indicators would be determined.

- The project actually employed the above compliance verification method and determined that the indicators were in fact implemented.

3.2.3 Implementations By Projects Surveyed

Three of the projects surveyed (Projects A, B, and D) deliberately structured their activities to produce pertinent and timely documentation. Of these three, all had rather informal standards for content and language, but the documents were reviewed by their intended audience. One of these projects (Project A), having produced a User Guide prior to Preliminary Design Review, utilized that document, once approved, as the foundation for testing. In

effect, for this project, the User Guide became a requirements baseline which shaped design and implementation. While documentation appropriate for the development in the other two projects (Projects C and E) was produced, it was accomplished at the end of integration testing, in the traditional fashion.

Three projects (Projects A, B, and D) made extensive use of Unit Development Folders (UDF), whose format, contents, and objectives were well defined. One of these three projects (Project A) found that the UDF concept was used most heavily during the early phases, and that once coding and implementation began their usefulness diminished. Two of the three projects instituted formal control, review and access mechanisms for UDFs, and continued to use the documents as management visibility tools beyond the design phase. For these projects, the UDFs continued to be used as repositories for evidence of the completion of coding, testing, and acceptance activities.

PRACTICE	PROJECT A	B	C	D	E
Document Pertinence					
Document Schedule	X	X		X	
Document Purpose	X	X		X	
Documents Phased	X	X		X	
Audience Review		X		X	
Content and Language Defined	X				
Control of Changes		X		X	
Procedures Stated	X	X		X	
Activity Performed	X	X		X	
Unit Development Folders					
Content Captured As Created	X	X		X	
Form and Content Established	X	X		X	
Contents Used	X	X		X	
Controlled Access		X		X	
Procedures Stated	X	X		X	
Activity Performed	X	X		X	

INDICATORS OF DOCUMENTATION STANDARDS BY PROJECT

FIGURE 3-4

3.3 DESIGN METHODOLOGY

Traditionally, software design has begun with a user's definition of requirements for a capability he needs. The first step is to identify the major functional components which will comprise the computing capability. The responsible programmer then proceeds to further partition each functional component into subroutines. The "kernels" are identified--those subroutines judged to be most difficult or critical to component function--and they are designed, coded, and compiled first. The programmer then proceeds to design, code, and compile subroutines immediately "above" or "below" the kernels within the component structure. This process is repeated until enough of the structure has been compiled to permit checkout of a group of routines. When all of the pieces of a component have been constructed and checked out in this fashion, the component is then handed over for integration with other components.

This approach evolved primarily in response to lengthy service times for keypunching and computer processing. Overlapping of design, code, and checkout activities was necessary to make effective use of the programmer's time. The strategy of developing the kernels first tended to minimize schedule risk, since as his completion milestone approached the programmer would have only the most straightforward pieces of a component left to finish. It has only been recently, with improved service turnaround--usually through the expedient of on-line terminals--that the programmer has been able to change his basic strategy.

Changes in the software procurement environment have occurred which have resulted in alterations in design practices; 1) there is a new emphasis on formalization of the user's requirements, 2) software developers are now obligated to perform a formal demonstration of the completed computing capability, and 3) the computing capability is often procured for a fixed price. Requirements are formalized early (no later than Preliminary Design Review), so that the customer can confirm that his needs are being adequately addressed, and the software developer can more precisely determine the costs to satisfy the requirements and more precisely control those costs. Faced with an obligation to conduct a formal demonstration of his product, the developer must confirm that the capability designed will meet the requirements specified. Further, to accomplish necessary demonstration test planning, he requires earlier visibility of details of the design than is traditionally possible. The fixed price contract environment has caused the developer to "tighten up" his design and testing activities; he can no longer afford to build "throw away" test drivers for lower-level code testing, and he no longer has the latitude to use the coding process to resolve and correct design errors and omissions.

Three new practices are being employed to accommodate these changes. The traditional development consisting of overlapped design, code, checkout and integration tasks, has been separated into two distinct activities: design, which culminates at the Critical Design Review, and construction (code, checkout and integration), which is initiated after CDR. There is a more formal and explicit design verification function included in the design effort, with the twin objectives of assuring that the design is workable

and that it will satisfy the requirements. Finally, the design activity itself is being performed in a deliberately top down fashion, from the functional component level to the level of "primitives" (i.e. library subroutines, operating system services, utilities, etc.). In the following subsections, these three Modern Programming Practices are discussed in detail.

3.3.1 Design Completion

The separation of software design from construction permits earlier visibility of the details of the design and allows more precise planning of the remaining construction and delivery activities. The design of the capability is developed until its level of detail throughout is sufficient for preparing a construction plan, and program specifications have been written to define each construction task. The completed design is documented, reviewed, and approved (at CDR) before coding and checkout begins.

This practice of completely defining the details of a software design before construction provides the visibility the Program Manager requires to plan the remainder of the development. With completion of the design document, the PM has a list of products (end items and their constituent parts) that he needs to precisely schedule construction activities. The PM can more accurately forecast resource expenditures and, armed with his knowledge, can more effectively monitor actual expenditures incurred. As the construction activities proceed, the list of products upon which the plan was based offer a means of assessing progress and determining completion.

Another implication of this practice is that personnel assigned design or construction tasks can be more specifically chosen on the basis of their skills and level of expertise. As discussed in Section 3.0, the Program Manager may not be able to have an individual assigned to him for the duration of the project. Because a component design task is a shorter assignment (than the traditional component development task which included design, code, checkout, and documentation) and because it now results in a specific end item (the documented, complete component design), there is another alternative. The Program Manager can negotiate for an individual to design a particular component, and retains the option to return that individual to his "home" organization, using another (less-skilled) individual to carry out component construction. This makes it possible for the PM to perform a better (and more cost effective) matching of skills to tasks.

It should be noted that the cost savings alluded to in the previous discussion are realizable in BCS only to the degree that personnel practices and labor agreements permit; these constraints may limit an individual PM's ability to plan task assignments, except as it is made necessary by the criticality and availability of skilled resources.

Our basic objective in investigating the five projects surveyed was to ascertain whether the software design was completed in accordance with its objectives before construction began, and whether the knowledge gained in the design activity was capitalized on in planning the construction activity. Accordingly, the questions we asked were designed to discover if:

- The project established a standard definition for design and program documentation.

- The project prepared a construction plan.

- Designs were formally reviewed prior to the start of code construction.

- The above review determined that the design was complete, at least in terms of the design document satisfying the standard definition and responding to all pertinent requirements.

3.3.2 Design Verification

As the design details of a software functional component are determined, and at the point when the design is considered complete, an explicit verification of the design can be performed; the mechanism for this is usually a peer review called a structured walkthrough. Such a review has as its basic objective verifying that requirements for a particular component have been adequately satisfied, and that the design itself is consistent. Beyond this basic objective, verification of design consistency may assume a more formal posture, in that reachability and modularity analyses of the design may be performed. This analysis can be performed manually as part of the structured walkthrough, or it may be performed with the aid of a computerized logic analyzer.

It must be acknowledged that traditional practices have often allowed the software developer to proceed through coding, integration, and even demonstration without realizing that some requirement was not implemented. The basic objective of this new practice is to prevent such oversights. Deliberate review of design against requirements may also point out less-costly implementations or reveal possible reductions of scope to meet cost targets; this helps ensure that the tasks planned for after Critical Design Review are those necessary and sufficient to implement the capability the customer requires.

Design verification is primarily a way of performing a portion of the traditional testing activity much earlier in the development process, i.e. before code is composed. In a project, design verification is often implemented by assigning the component designers and the test designers the responsibility of reviewing each other's work.

Design verification minimizes the need for later code rework to solve errors and omissions in the design. Uncovering design errors at this stage tends to be less costly, because fewer end items must be changed to implement a solution. Schedule risk is reduced by minimizing the need for code rework, which would ordinarily occur late in component checkout, or even later during integration.

An additional benefit of design verification is that it may reveal that the requirements, as originally stated, were mis-prioritized. In attempts to trim the cost to a fixed price, the developer (with the customer's concurrence) may have deleted a requirement which was crucial to successful operation, or design verification may reveal that a requirement was simply

missed during Definition. Armed with this information at Critical Design Review, the developer and customer can agree to 1) revise the requirement priorities to get a working capability and still stay within the fixed price, or 2) accept the out-of-scope requirement at additional cost. This kind of visibility permits the customer to take needed remedial action in a timely fashion (e.g. revise his plans for using the redefined capability, or secure additional funding).

A final observation must be made about design verification. Requiring review of one individual's design by another implies that a formal method to represent the design, the rules of which are understood by both the designer and the reviewer, should be established as a project standard. The traditional design representation is the flowchart; newer representation techniques such as HIPO (Hierarchical Input-Process-Output) diagrams can also be used for this purpose. The important point is not which specific design representation technique is used, but rather that one be used consistently on the project and serve as an effective communication aid in the verification of the design.

We were interested in determining on the projects we investigated 1) if the practice of design verification was adopted, 2) how it was implemented, 3) what the objectives were and what types of analyses were performed during the walkthroughs, and 4) if the practice was consistently used throughout the design activity. Our questions were devised to determine if:

- The project established procedures which specified the use of design verifications, the conduct of such reviews, the review participants and their responsibilities, and the compliance evidence to be produced as a result of such reviews.

- The project established procedures which stipulated the use of design reachability and modularity analyses, the method to be used to perform the analyses, and the compliance evidence to be produced as a result.

- The project established the method by which compliance with the above procedures would be determined.

- The project actually employed the above compliance verification method and determined that the procedures were in fact implemented.

3.3.3 Top Down Design

Top down design is advocated by its proponents as a means of making it possible, throughout the design activity, for the designer to test assertions 1) that the design is correct (i.e. that all requirements are satisfied), and 2) that the design will work (i.e. that it is complete and logically consistent). Top down design is accomplished by decomposing the identified requirements level-by-level, and by constituent parts across levels.

In traditional practice, the first decomposition of requirements occurs when a set of (one or more) functional components is defined, which together will provide the necessary capability. From that point on, however, the

traditional approach has been to detail the individual component designs in terms of "subroutine-able" functions which could be linked together to provide the required component capabilities. For several reasons, this approach has often resulted in designs which 1) failed to satisfy all of the requirements, (and) or 2) included capabilities not required by the customer.

The top down approach simply requires that the details of individual component designs be accomplished in essentially the same manner as the initial decomposition. That is, the requirements for a component (and, at lower levels, for a constituent part of the component) are decomposed to specify a complete set of next-level elements that will provide the required capability. This specification is then tested against the assertions of correctness and workability before proceding to decompose each of the next-level elements. Thus, at each level in the design, a complete and explicit apportionment of the customer's requirements is developed.

This top down approach eliminates the costs associated (traditionally) with designing, coding, checking out, and integrating software elements that implement functions not required by the customer. In addition, the top down approach assures that the software developer does not have to bear the (often considerable) costs of "adding in" software elements at a later point in development to implement previously unsatisfied or ambiguous requirements.

Additional benefits from the top down design approach can be realized in checkout (which can also proceed top down), in design verification (which can avoid propagation of design errors), and in more deliberate use of existing support library routines.

Top down checkout has been recognized traditionally as a cost-effective approach, primarily because special test drivers are not required to exercise groups of subroutines. In traditional practice, however, the "kernels first" approach meant that test drivers were developed, or that testing was deferred until enough of a component's structure was built to permit a top down exercise of the code. In the modern software development environment, deliberate planning of (checkout and integration) test activities occurs in parallel with component design. The top down design approach permits more concise test objectives to be formulated; the test designer has the earliest possible visibility of precisely what must be done to test and integrate the pieces of the software capability. Test planning can therefore proceed in step with the design process itself, and can be completed early enough to permit timely checkout and integration.

The performance of design verification reviews at each level of design refinement allows logic errors in the design to be detected before they are propagated throughout lower design levels or into the code itself. Successive structured walkthroughs of the design as it proceeds increases the costs directly associated with the design activity, but these costs should be more than offset by reducing the costs associated with resolving, in the code, design errors discovered during checkout and integration.

By constructing a component's design in a top down fashion, software available in support libraries (e.g. standard subroutine libraries, operating system services, utility programs) can be more widely and efficiently used. In the traditional "kernels first" approach, the design of the kernels often precluded the use of off-the-shelf routines immediately above or below the kernels in the component structure, because interfaces were incompatible or requisite capabilities were not supplied. In the modern environment, where designing to a fixed cost is a prime motivator, components are designed to take maximum advantage of available support. The cost savings inherent in designing to intentionally use existing software routines are increased further by savings in coding and testing.

The top down design is completed to a level of detail that is sufficient for coding. At each design level, the "primitives" (functions directly implemented in the programming language, or support library routines to be used) are identified.

In examining the selected projects for evidence of implementation of this top down design discipline, we were particularly interested in discovering if the design process permitted design verification to occur. We therefore structured our questions to discern evidence of a standard and consistent design representation technique. Our inquiry was designed to determine if:

- The project specified a deliberate top down design approach.
- The project specified standards for representing the design (e.g. subroutine trees, flowcharts, HIPO diagrams, etc.).
- The project specified criteria for design completion.
- The project specified methods of refining the design to accomodate support libraries available.
- The project consistently applied all of these techniques, as evidenced by the material available in their design document(s).

3.3.4 Implementations By Projects Surveyed

Three of the projects (Project A, B, and D) completed the software design and submitted it for evaluation at Critical Design Review before construction began. In these cases, the CDR was phased; as major "branches" of the system design tree were completed to the primitive level, those branches were reviewed and signed off for the start of coding. Two of the projects (Project B and D) established a code construction plan as a result of the design activity.

Three of the projects (Projects B, D, and E) employed structured walkthroughs of component designs. All three projects felt that the most effective way to conduct a structured walkthrough was in a "one-on-one" environment, where the designer presented his component to a peer, a lead programmer, or a technical customer representative. The projects relied on the structured walkthrough mechanism to analyze component designs for stipulated reachability and modularity characteristics. A fourth project (Project C) employed

structured walkthroughs only during the initial stages of preliminary design.

All of the surveyed projects established standard definitions for design and program documentation; three of the five (Projects A, B, and D) reviewed the documentation for satisfaction of stipulated standards prior to code construction, while the other two (since the documents were not completed until the later testing phases) reviewed them immediately prior to delivery of the system.

All of the five projects surveyed used a consistent and formalized top down design approach. Such techniques as subroutine trees and HIPO diagrams were used to represent the design, during both the Definition and Design phases. Design completeness criteria were usually not formally specified or adhered to. One project (Project B) developed a "working definition" of completeness which they felt was a good yardstick; namely, that the design was considered complete if it had been carried down to the level of primitives with no intervening decision points ("IF" statements).

	PROJECT				
PRACTICE	A	B	C	D	E
Design Completion					
Standard Documentation	X	X	X	X	X
Construction Plan		X		X	
Review Prior to Code	X	X		X	
Procedures Stated	X	X		X	X
Activity Performed	X	X		X	X
Design Verification					
Structured Walkthroughs		X	X	X	X
Objectives Stated		X		X	
Reachability and Connectivity		X	X	X	X
Modularity Analysis		X	X	X	X
Control of Verification		X		X	
Procedures Stated		X		X	
Activity Performed		X		X	
Top Down Design					
Top Down	X	X	X	X	X
Design Representation	X	X	X	X	X
Design Refinements	X	X	X	X	X
Design Completeness Criteria		X		X	
Procedures Stated	X	X	X	X	X
Activity Performed	X	X	X	X	X

INDICATORS OF DESIGN METHODOLOGY BY PROJECT

FIGURE 3-5

3.4 PROGRAMMING STANDARDS

Software developers have typically established standards for coding to maximize productivity (by specifying a higher-order language), to ease readability (by requiring commentary in source listings), and to simplify linking of routines and integration testing (by establishing interface conventions). However, higher-order language standards are frequently circumvented--in the interests of achieving greater machine efficiency--and the inclusion of appropriate comments into the code is neglected when pressures are high to speed the code production process. Debug aids such as dumps and traces usually depend on adherence to established linkage conventions. These particular programming standards are ordinarily observed, if only to avoid increasing the costs of testing.

Some changes in the environment in which software is produced are encouraging adherence to the other traditional programming standards. With improvements in language compilers and operating systems, the programmer has much less need to resort to assembly language; instead of having to develop his own interrupt handlers, input/output drivers, etc., the developer can use standard system functions. There have also been substantial hardware improvements; much faster central processors, larger main and auxiliary storage, and even operating system features which make mainframe memory appear (to the application program) larger than it actually is.

Substantially less of the code now prepared to implement a capability is directly involved with the functions of task/storage/data management, and with input/output hardware control. These functions are now accomplished by "system" software, and invoked in the application software code by relatively simple statements. The programmer now produces code which is much more oriented to the functions of the user's application than to interfacing with (and operating) the computing hardware.

A further implication of this change is to place more emphasis on the application for which software is being built and less on the internal workings of the computer. In the past, to obtain help with what was then the larger part of his job, the application programmer required ready access to the hardware and systems specialists. For a variety of reasons, this access was frequently difficult and help, when provided, was often not pertinent to the application. This meant that the programmer might rewrite the same piece of software several times until he arrived at something which would work in its hardware environment.

Today, the larger part of the programmer's task involves the functions of the user's application. The customer will often go out of his way to have his people readily available to advise and assist with the functions the programmer is implementing. Perhaps even more important, these advisers have a vital interest in ensuring a successful implementation.

Another change which has had significant impact on the application of programming standards is the increasing sophistication of the software customer. The customer is procuring software capabilities with fixed-price, competitive solicitations. This makes it virtually imperative for the software developer to use productivity aids such as higher-order languages. The developer must

stress incorporation of available "standard" software, to avoid incurring the costs of "re-inventing the wheel." The capability and reliability of "standard" software has reached a level such that it is almost always cost-effective to employ it; further, using "standard" software in an application requires strict adherence to established linkage conventions in the software written to interact with it.

The more sophisticated customer is less tolerant of "coding errors" than in the past; he expects the developer to "do it right the first time." The customer is not as willing to accept a software capability with known discrepancies, omissions, or errors, and allow the developer to fix them at the customer's expense during maintenance. Rather, the customer is insisting that these shortfalls be corrected before he will accept the product, and at the developer's expense. Perhaps most important, any member of the software industry competing in the commercial market depends upon a reputation for delivering a quality product, and is loathe to deliver computer programs with known deficiencies.

At BCS, three Modern Programming Practices are being used to improve the quality of the code produced: structured forms, coding conventions, and code verification. These practices are discussed in the following subsections.

3.4.1 Structured Forms

The use of structured forms in writing source code (IF-THEN-ELSE, DO-WHILE, etc.) has been adopted primarily because the forms correspond more naturally to human thinking processes than many of the traditional formulations of software logic. In addition, these coding constructs lend themselves fairly readily to design representation. A closer correlation between code and design means that fewer "coding errors" result. Further, use of structured forms frequently results in simpler, blocked coding sequences, fewer total source code statements, and fewer total possibilities for error. Structured forms make the source code easier to review, so that such errors can be more easily recognized and corrected.

As mentioned in Section 2, the traditional obligation of the software developer is to provide the logic structure that surrounds the client's algorithms. That obligation remains unchanged; the primary benefit of structured forms is the lessened likelihood of an error in the structure.

To determine that a project had consistently applied structured coding forms, we posed our questions to ascertain whether:

- The project established (perhaps by reference) standards of acceptable logic constructs to be used in source code.

- The project determined what logic constructs (supported by the programming language) would be prohibited (e.g. unconditional GO-TO).

- The project established compliance mechanisms and a means of handling exceptions.

3.4.2 Coding Conventions

Programming standards specifying the use of a higher-order language and the inclusion of meaningful commentary are traditional in software development. While recognizing the value of these standards, the programmer has, until recently, found them difficult to follow. As mentioned earlier, it is easier now for the programmer to use a higher-order formulation throughout the application; he does not have to lapse into machine code to perform a specific function, because that function is usually supported by the compiler. In addition, language and commentary standards are now being established as acceptance criteria, enforced by the customer to give him latitude to repair and enhance the delivered code himself, or to contract with a third party for that function.

New naming conventions have been found necessary to make cross-referencing among software items and related documentation easier. The project requires that names used in the code to identify functional units and data be both more descriptive (which is now possible because of relaxed compiler restrictions on name length and format) and correspond more closely to the symbolic names used in the design representation.

Briefly, we interviewed our projects to establish the existence and application of:

- Standard definitions for interfaces such as calling sequences, external data accesses and error processing.

- Naming conventions for units of code and data, which make them relatable to other software and to supporting items.

- Standards for code organization and generation of comments in the code.

3.4.3 Code Verification

In addition to verification of design, as discussed in Section 3.3, verification (by review) of code is employed to assist the construction (coding and checkout) activity. As each subroutine is coded, a peer review or inspection of the code is performed to assure that it is ready for checkout. This peer review assists the programmer in finding errors he might otherwise miss, and reduces the number of computer runs he requires to checkout a subroutine.

Once the programmer has completed checkout of his functional component, another review may be performed to insure that the code is ready for integration testing. The programmer submits evidence of successful completion of his checkout, and describes the logical and functional testing he has employed. This review evaluates the adequacy of these component tests and provides a way of conclusively closing out the construction task.

In gathering data for this study, we were interested in whether a deliberate review or inspection occurred during construction, or at least at its conclusion. The projects we interviewed were asked if:

- Peer reviews of compiled code were planned, with procedures defining participants, evaluation criteria, and evidence of review completion.

- Reviews were planned immediately prior to component hand over for integration, to verify the adequacy of checkout testing and results.

- These reviews were actually conducted as evidenced by tangible review results, and resulting action items were resolved where necessary.

3.4.4 Implementations By Projects Surveyed

All five of the projects used structured forms consistently throughout their implementations; one of them (Project A) did not explicitly prescribe permissible structured forms and verify compliance with those forms, although that project felt that the source language they used did not contradict the intent of structured forms inherent in their detailed designs. Three of the projects (Project B, C and E) deliberately avoided the use of the unconditional GO-TO.

While all of the projects established naming conventions for blocks of code, subroutines, and programs, in two cases (Projects B and C) similar conventions for data variable names were not implemented. Three of the projects (Projects A, B, and D) established uniform methods of defining interfaces such as calling sequences, external data, and error conditions. These same three projects instituted standardized formats for code organization and commentary. The explicit objective of these standards was to ensure that the code created was easily relatable to the documented design. In most cases, symbolic names (relatable to the design document) and a short statement of the function of each block were the minimal standard commentary.

Two of the projects (Project B and D) instituted deliberate code verification procedures in the form of peer code reviews; however, only one of these (Project D) attempted to continue the practice throughout the construction activity, and it met with only partial success. Both projects having experience with this practice felt that it was extremely expensive, and that it was difficult to motivate project personnel to consistently apply this technique. Furthermore, the projects observed that as the pressure of deadlines increased, this review technique was virtually impossible to employ. The same two projects did keep documented Unit Development Folder checkout records.

PRACTICE	PROJECT A	B	C	D	E
Structured Forms					
Logic Statements Defined		X	X	X	X
Compliance Established		X	X	X	X
Procedures Stated	X	X	X	X	
Activity Performed	X	X	X	X	X
Coding Conventions					
Syntactical Forms Allowed		X	X	X	X
Interface Conventions	X	X		X	
Naming Conventions	X	X	X	X	X
Code Organization and Comments	X	X		X	
Procedures Stated			X		X
Activity Performed			X		
Code Verification					
Checkout Documented		X		X	
Peer Reviews		X		X	
Procedures Stated		X		X	
Activity Performed				X	

INDICATORS OF PROGRAMMING STANDARDS BY PROJECT

FIGURE 3-6

3.5 SUPPORT LIBRARIES AND FACILITIES

It is common practice for software developers to incorporate "standard" elements of software (e.g. subroutine libraries, operating system services, utility programs) into the software capability they are responsible for providing. Use of "standard" software elements has been practiced traditionally to reduce the costs of re-developing code for common functions, and to improve the reliability of the software capability being produced. Further, it has become virtually imperative that the software developer employ these elements, because of the increasing sophistication of operating systems and hardware. An important objective of this practice is to protect the integrity and, hence, the reliability of the facility on which the developer's software will operate.

The facility characteristics of reliability and integrity are realized, in practice, only on "matured" computer systems. It should be noted that the recent tendency among vendors has been to build new products (hardware and system software) which are less-than-radical departures from that vendor's predecessor systems, thus retaining "matured" elements as much as possible. Also, recent improvements in higher-order languages and the software libraries and facilities which support them have further reduced the programmer's need to resort to special techniques to achieve sizing and timing requirements, or to avail himself of special hardware or system software features.

Even when a new or specialized computer is to be the host for an application, the software developer need not do without "standard" elements. The design of (a subset of) some existing mature operating system can be implemented on the host, and cross-compilers can be used to produce libraries for it by copying software from another, proven system.

In essence, then, traditional practices of using available, standard software whenever possible have become common and even imperative in the modern environment. There are now, some new opportunities for employing system-like support libraries and facilities in software development work.

The modern design practices employed in software projects encourage the establishment of (project-specific) standards for design representations. These representations contain much more design detail, and the need to consistently express these details places an increasing burden on the software designer. The more rigorous design verification activity requires that many more design details be analyzed and inspected. Manual design verification can be a significant consumer of (human) resources if it is done rigorously, and it is difficult to ensure that the reviewer has in fact been thorough in his verification. New automated design aids are evolving to facilitate design representation and to assist in design verification.

The new emphasis on use of structured forms for expressing software logic has caused some difficulties in practice. Since language compilers currently available do not directly support (all of) these forms, the programmer must define alternate forms, acceptable to his compiler, which meet the intent of structured forms. In addition, he must avoid using compiler-supported forms which violate the intent of structured logic. In practice, projects must often resort to manual inspections to ensure that such coding standards

are followed. These inspections are also being used to establish that the code faithfully corresponds to the documented design. The inspection process has proven to be costly and only partially effective. New structured precompilers, which directly support structured logic forms and perform logic correctness and consistency checks, are being developed to reduce the software developer's reliance on inspections.

New management practices focus on tracking the tangible results of the software development process in order to manage that process more effectively and to control costs. In practice, even for small projects, the sheer number of individual items produced (for each subroutine, a design specification, source code, object code, compiler listing, test data, etc.) is such that a significant clerical burden is imposed. To assess status, the Program Manager needs to know which items have actually been produced and signed off. To assess progress, the Program Manager needs to know what items were planned for completion on a specific date, and whether those items have in fact been completed on schedule. To assess performance, the Program Manager needs to know how expenditures experienced in producing on-going and completed items compares with his plan. Again, the amount of detailed information required to make those assessments precisely makes this practice extraordinarily difficult and costly to implement solely with manual methods. Programming support library aids are being developed which support the recording and controlling of coded items (designs, code, test data, etc.) and assist in modifying these elements. These aids are also providing the visibility needed by management for control purposes.

The automated aids being developed within the industry have as their specific objective reducing the burden of repetitive, detailed tasks that has hindered the implementation of MPP. At BCS, we have developed several such tools and made them available for project use; these tools will be described in this subsection.

It should be noted that, while the aids developed by BCS are representative of the current state of the art in the software industry, it is evident that much remains to be done to fully realize the potential for using the computer to make software design, construction, and control activities more cost effective. In particular, when we surveyed the projects for this study, we found considerable dissatisfaction with the current state of automated aids for Modern Programming Practices, particularly when these aids were compared with traditional (and more mature) support libraries and facilities. The shortfall of newer aids in adequately meeting the real needs discussed above was so apparent that we found it necessary to give equal consideration in our study to the projects' use of manual alternatives in assessing the formality of their MPP implementations. Therefore, as we describe the evidence we looked for in each case, that evidence will include definitions of the manual practices and procedures used in conjunction with (or in lieu of) the available automated aids.

3.5.1 Design Aid

Automated design aids facilitate the recording of design representations, and perform basic checks for logical consistency in the design. The BCS

design aid used by some of the projects we surveyed accepts a stylized formulation of a software design, and translates this into a "standard" design representation. Specifically, the representation can express (and aid in the communication of) the function of a process (typically, a subroutine), its interfaces with other processes, and the conditions, causes, and effects for transitioning between subprocesses within it.

This same design aid helps verify whether the design expressed is complete and consistent. In particular, it can diagnose whether all possible conditions are accounted for at transition points, whether the design uses certain prohibited transitions (jumping into the middle of a loop, for instance), and whether there exist conditions for which the process would not terminate with a normal exit.

Manual aids supporting the design task include document prototypes, procedural forms, and design analysis algorithms. A document prototype describes the expected format and content of a portion of documentation to aid in its preparation. Forms may be used to help conduct and record the results of design verification via structured walkthroughs. An analysis algorithm can aid in transforming a design representation into a form that uses only simple control logic.

Since the use of automated or manual design aids is predicated upon a defined and consistent design language, we solicited from our projects evidence of such a defined standard. We also asked if the BCS automated aid was used, and whether manual techniques were employed in support of (or in lieu of) the automated aid. Specifically, our inquiry attempted to establish whether:

- The project defined a "standard" design language.
- The project established procedures for using that language to develop, document, and verify designs.
- The project used the BCS automated aid to facilitate compliance with their procedures.
- The project used manual aids (document prototypes, forms, analysis algorithms) to facilitate compliance with their procedures.
- The project consistently applied their standards and procedures, as evidenced in available design documentation.

3.5.2 Structured Precompiler

Structured code precompilers translate structured logic forms into formulations acceptable to "standard" compilers, and check for closure and nesting consistency of the code structures. The BCS structured precompiler performs this function for programs coded in the Fortran language. Besides its capability to process common structures (IF-THEN-ELSE, DO-WHILE, etc.), this particular aid accepts an enriched set of structured formulations and provides options for statement identification and block identification to increase code readability.

The BCS structured precompiler provides only limited support to code verification. Specifically, it is capable of checking for closure of logic blocks and verifying certain other details of the formulation, such as nesting consistency.

In the absence of a precompiler which supports the higher-order language in use on a project, the project can establish standard logic formulations which are directly supported by the compiler and meet the intent of structured forms. In addition, the project can prohibit the use of compiler-supported formulations (such as the unconditional GO-TO) which violate the intent of structured logic. To ensure these standards and prohibitions are observed, the project can employ manual code inspections and require signoff of coded elements.

Since the use of a structured precompiler or manual code inspections is predicated upon a definition of acceptable logic forms, we solicited from our projects evidence of such a standard. We also asked if the BCS structured precompiler was used (on those projects which were producing Fortran code). For those projects which were not employing the precompiler, we asked how compliance with structured logic forms was assessed. Our inquiry attempted to establish whether:

- The project identified structured logic forms (and prohibited certain other forms) to be used in producing code.

- The project employed a structured precompiler which supported these logic forms, or

- The project established procedures for inspection of the code to assure that structured logic forms were used consistently.

- There was tangible evidence (precompiler listings or inspection signatures) of consistent adherence to the defined standards.

3.5.3 Programming Support Library Aids

A programming support library aid is used to record code (and computer-readable design formulations), data, and other card-type statements used in software design, construction, and testing. This type of aid assists in modifying recorded elements and performing related file maintenance services. In addition, this type of aid can provide the visibility needed by management for control purposes.

The BCS programming support library aid is capable of recording most of the common types of products of the programming process--source code, object code, data cases, job control language procedures, and even the design-expression and macro-language statements used as input to the two previously-cited aids.

This aid provides capability to modify (delete, add, update) recorded elements, and to perform several other file-maintenance services commonly needed in the process of producing software.

This aid provides rudimentary support for controlling and tracking the item-results of programming. In particular, it is capable of listing the identifications and dates-of-entry of the elements stored, but provides no other assistance in matching this list to a program plan or schedule.

If a programming support library aid is not employed on a project, the project can take advantage of (traditionally available) text-editing, data dictionary, and copy library capabilities. These tools do not offer the comprehensive control and support of a programming support library aid, but can assist in recording and modifying code elements. When these tools are used in conjunction with manually-maintained ledgers (identifying the names, contents, locations, versions, and relationships of code elements), visibility for management control can be provided. Where even these traditional tools are not available, the project must rely on manual techniques to develop, capture, and modify code elements, as well as to provide control visibility.

In our investigation of projects for this study, we were interested in learning if the BCS programming support library aid or other automated tools which facilitate recording and modifying code were employed. In addition, we asked whether manual ledgers or records were used in conjunction with the aid or tools to provide management visibility and control. Our questions were structured to determine whether:

- A controlled library of code elements was used by the programmers to facilitate construction and modification tasks.

- An automated aid (or tools) was used to create, maintain, and provide access to the controlled library.

- Manual forms and procedures were used (perhaps in conjunction with the automated tools) to provide management visibility of the status of the controlled library.

3.5.4 Implementations by Projects Surveyed

All of the projects utilized a formalized top down design language, supported by such manual aids as design forms and HIPO charts. One of the projects (Project D) successfully developed its own project-specific tool for expressing and verifying designs at the system and functional level. Two of the projects (Project B and E) employed the available BCS design aid in design documentation and verification. Both of these projects used this aid only during the component specification activity, and felt that this particular package was both clumsy and expensive in implementation, and so rudimentary in its design verification as to be essentially useless beyond that point; its use was discontinued.

One of the projects (Project C) used the BCS-developed precompiler in the construction of code. Two of the projects (Projects B and D) instituted inspections of the code to assure that prescribed structured forms were adhered to.

Three projects (Projects A, B, and D) used vendor-supplied text editing,

data dictionary, and copy library capabilities. Two of these three (Projects B and D) kept manual records of controlled libraries for management visibility. Another project (Project C) instituted item control during functional testing (see Figure 3-2), and used the BCS programming support library aid. This project felt that the support afforded by the aid was critical to the successful packaging, delivery, and acceptance of the package, even though control was not established until quite late in the development lifecycle. In particular, since the end product involved consisted of several tailored configurations, the aid, in conjunction with elaborate manual procedures, supported the configuration distribution activity quite well.

PRACTICE	PROJECT A	B	C	D	E
Design Aids					
Design Language Standards	X	X	X	X	X
Manual Aids	X		X	X	X
Automated Aid		X		X	X
Procedures Stated	X	X		X	
Activity Performed	X	X		X	
Structured Precompiler					
Structured Programming Standards	X	X	X	X	X
Inspection		X		X	
Automated Aid			X		
Procedures Stated		X	X	X	
Activity Performed		X	X	X	
Programming Support Library Aids					
Controlled Libraries	X	X	X	X	
Manual Aids		X	X	X	
Automated Aids	X	X	X	X	
Procedures Stated	X	X	X	X	
Activity Performed		X	X	X	

INDICATORS OF SUPPORT LIBRARIES AND FACILITIES
BY PROJECT

FIGURE 3-7

3.6 TESTING METHODOLOGY

The developer's traditional approach to software testing has concentrated on the operability and logical integrity of the code. The objectives of testing have been to confirm that the components of a capability perform successfully together, and that the capability "does not unexpectedly halt, loop, or exit."

In this traditional environment, the customer retains the responsibility for the functional integrity of the capability, i.e. the responsibility for determining that the answers produced by the algorithms implemented in the software are correct. This is appropriate: the algorithms originated with the customer, and often involve principles (engineering, financial, etc.) outside the software developer's knowledge, training, and experience. The software developer's obligation is to assure that the customer's algorithm was translated into computer code without error; typically, the programmer checks the translation of algorithms to code by hand-calculation. This kind of checking does not verify the correctness of the algorithm itself, but only that the algorithm specified was faithfully implemented.

Previous discussions in this section have noted several changes in the software procurement environment and how they affect the software developer's management, documentation, and design practices. These changes also significantly impact the approach to software testing. Specifically, current procurements are requiring that the software developer demonstrate, at the time of delivery, that the package (software, documentation, procedures, data, etc.) that has been produced adequately meets the customer's needs.

Formal demonstration of the software capability (also called acceptance test) has the objective of confirming that the complete package, as delivered, is ready for immediate operational useage. To ensure this, the customer usually insists that an exercise of his own devising, typical of his intended use and employing his own "live" data, be part of the demonstration.

To assure that this useability exercise proceeds smoothly, the software developer requests that the customer's exercise be made available prior to the acceptance test, so the developer can conduct a "dress rehearsal" of the acceptance test at his own site.

The customer-provided test case is typically not comprehensive (i.e. does not demonstrate every function of the capability), but rather is intended to demonstrate the functions which the customer feels will be "most commonly used," in the manner in which he intends to use them. Since the software developer has no advance knowledge of which functions the customer's exercise will employ, he must assure himself beforehand that all required functions of the software have been verified to work as designed. To accomplish this, the software developer subjects the software system to a comprehensive set of functional tests.

The practices of formal acceptance demonstration and comprehensive functional testing can be expected to increase the costs attributable to software development, since functional testing has traditionally been conducted by the customer following delivery, at his own expense. In the interests

of keeping this increase to a minimum, the software developer places new emphasis on formal test planning and preparation. The objectives of more rigorous test planning are to ensure that all tests planned are necessary and sufficient to prove that the developed capability is complete, correct, and operable.

The Modern Programming Practices of acceptance testing, functional testing, and test formalism are discussed in the following paragraphs.

3.6.1 Acceptance Testing

The primary objectives of acceptance testing are to confirm that the delivered capability is ready for operational useage and to show that the software developer has satisfied his contractual obligation.

To prepare for this formal demonstration, the software developer identifies the customer's acceptance test requirements, and prepares and submits an acceptance test plan for customer concurrence. Typically, the formal demonstration plan will include tests for installability, operability, and useability. The test for installability involves actually installing the software on the customer's facility, using installation procedures and materials previously prepared. The test for operability is performed after the capability has been installed, to demonstrate that the software will execute and produce results on the customer's machine. The useability tests include the exercises submitted by the customer, and demonstrate that the software will perform functions which the customer feels are typical of his intended use.

To insure that the formal on-site demonstration is an orderly process, the software developer typically conducts a "dress rehearsal" of the demonstration at his own facility, using the acceptance test procedure (and materials he has prepared) to perform the installability, operability, and useability tests, including the useability exercise devised by the customer. This dress rehearsal is conducted under conditions which are as close as practical to those which will exist at the customer's facility (i.e. the same procedures, materials, hardware configuration, and even individuals with the same skills as the intended users). The software developer packages the results of this dress rehearsal into a set of "control listings" which can then be used to verify that the results obtained during demonstration at the delivery site are as expected.

The dress rehearsal and formal demonstration activities are part of what has traditionally been called System Test. These new practices are employed to avoid what in the past has been a serious problem, both for the customer and the software developer; namely, that the user has not been provided at delivery all the materials he needs for operational usage. This situation is damaging to the reputation of the developer, and expensive for the customer, since he must bear the additional costs of completing the package for use in its intended environment.

The identification of acceptance test requirements and the preparation of a plan for acceptance testing are activities performed by the software developer prior to Critical Design Review. At CDR, this plan (identifying

tests for installability, operability, and useability) is presented to the customer for his concurrence. After CDR, the developer prepares the test procedures and materials for all tests, and the test data to support the installability and operability tests. At PCA/FCA, the developer requests the test exercises which the customer has prepared, so that he can complete preparation for formal demonstration.

In our investigation of projects for this study, we were primarily interested whether documented acceptance test requirements, plans, procedures, and supporting materials were in existence. In addition, we asked for evidence that a dress rehearsal was conducted prior to formal demonstration. Finally, we asked if the formal demonstration was conducted according to the acceptance test procedure. Our inquiry was intended to determine whether:

- The customer's acceptance test requirements were identified.
- A documented acceptance test plan and test procedures were prepared and reviewed with the customer to ensure that the acceptance test requirements were satisfied.
- A readiness review (PCA/FCA) was conducted to ensure that the capability was complete and that all functions of the capability had been verified.
- The project conducted a dress rehearsal of the acceptance test in an environment as close as practical to the intended using environment.
- The project committed to a date when the formal demonstration would occur and used the control listings produced in the dress rehearsal as evidence of readiness for the demonstration.
- The formal demonstration was conducted according to the acceptance procedure and the customer signed off on the tests, with exceptions or deviations noted as appropriate.

3.6.2 Functional Testing

As mentioned earlier, the software developer usually has no advance knowledge of which functions the customer's useability exercise will test during formal demonstration. The objectives of functional testing are to verify all the functional capabilities work as designed, and to subject the software functions to the test of reasonableness. Functional testing, in practice, may occur in lieu of or in parallel with integration testing; evidence that all functions have been verified is presented as satisfaction of the Functional Configuration Audit (FCA) objectives of PCA/FCA.

Functional testing is performed according to documented plans and procedures, and may employ "live" data supplied by the customer. The functional test plan may be devised in consonance with the integration plan; that is, the planned tests exercise the various software functions in an order consistent with the submission of individual components for integration testing. Functional testing may be conducted by the integrators, by an independent functional test team, or even by the customer himself. If individuals other than the

integrators perform the functional tests, they are not authorized to make changes to the software under test. Instead, a method whereby the functional test team reports problems which occur during testing is established, and the integrators (or programmers) are responsible for responding to those problem reports.

The customer's involvement (if not his direct participation) in functional testing is important. The customer participates in test planning and in the examination of results. The software developer cannot, with his background and skills, readily assess the reasonableness of the results of functional test. Instead of rigorously confirming the adherence of the coded algorithm to the customer's specification, the software developer submits the test results to the customer, who can usually determine by inspection if they are reasonable for the given input data. If the results are not reasonable, then 1) the customer can examine his algorithm specification to determine if it is correct, and 2) the software developer can examine his software for a coding error.

To ensure that functional testing proceeds in an orderly way, the software developer defines a functional test plan which identifies the type of tests to be performed and the data and materials required to support these tests. This plan is submitted to the customer at CDR for his review and concurrence. The customer may provide his own "live" data suitable to support several of the planned tests. After CDR, the software developer produces the test procedures and other supporting materials (which may incorporate customer-supplied data). After functional tests are conducted, the customer reviews the results for reasonableness, and assists in diagnosing problems which may be discovered.

In examining the software projects selected for this study, we were primarily interested in the formality with which functional testing was performed. We determined that there were several techniques which a project could employ to formalize the functional test activity: instituting a hand over and inspection of the software to be tested, chartering an independent team to perform the testing, defining a method for reporting and resolving problems discovered during functional test. Specifically, our inquiries were intended to determine whether:

- The objective of functional testing was to ensure that the delivered code satisfied all identified functional requirements.
- Functional test plans, procedures, and data cases were developed to satisfy this objective, including pass/fail criteria.
- The project chartered an independent test team or had direct customer participation in functional test conduct.
- The programmers submitted the software for inspection and hand over into a controlled library for the use of the testers.

3.6.3 Test Formalism

The objectives of test formalism are to ensure that all tests planned are necessary and sufficient, and that their results conclusively prove that the developed capability is complete, correct, and operable. The test specifications identify all the activities required, such as preparing test procedures and test data cases, and determining expected results. Resource requirements for each activity are estimated, and specific task assignments are detailed; these become part of the Program Manager's overall project plan.

Formal test planning is employed to reduce the costs of testing; tests are deliberately planned to take advantage of "live" customer data where possible, and are sequenced to minimize resource expenditures for building test drivers. To ensure repeatability of tests, test procedures and data are placed in a controlled library.

The customer's increasing involvement in design verification and test planning activities permits early validation of algorithms to be incorporated in the code, and allows the customer's "live" data and expected results to be employed both in design verification and functional testing. In addition, the "live" data cases can be used to produce realistic illustrations which can be directly incorporated into User Guide, training materials, and other documentation.

The strategy of planning individual test tasks to reduce the need for creation of specialized test drivers has been difficult to apply in conjunction with the "kernels first" traditional design approach. The practice of top down design development and verification in a level-by-level fashion allows this test strategy to be realized in a more natural way; that is, testing can now actually proceed top down, reducing the need for drivers to explicitly test lower-level software elements.

Traditionally, test data is used in one set of tests and then modified to create conditions for the next set of tests. The errors discovered in the first set of tests are corrected and changes introduced into the software must then be re-tested to assure the original problems were resolved. In traditional practice, however, recreating the original test conditions is often costly and difficult since the test data itself has been modified. In the interests of being able to recreate tests (to verify that code changes actually fixed a problem previously encountered) in a cost-effective fashion, the test materials (e.g. test procedures and data) are now constructed and controlled in a more disciplined way.

The software developer identifies at Preliminary Design Review the method (e.g. inspection, analysis, simulation, exercise, demonstration) by which each required functional capability will be tested. After securing customer concurrence at PDR on the test methods, the software developer produces a test plan, which identifies (or names) each specific test which will be conducted.

The developer then proceeds into test design. Given the test method, a design

for each test is prepared, which identifies each exercise within the test, relates each exercise to the function(s) it will test, and describes the sequence of exercises. The test design also includes a definition of the type of data and materials required to support each exercise. The software developer also devises a detailed schedule for the construction of test procedures, data, and materials, and for the actual conduct of tests. This schedule is integrated with the Program Manager's construction plan. Test designs and the test schedule are presented at Critical Design Review for customer concurrence, and to solicit customer participation in testing (e.g. providing test data, reviewing test results).

In our project inquiry, we were primarily interested in the formality and detail with which tests were planned. In addition, we attempted to discern the manner in which the tests were conducted, i.e. if the pass/fail status of individual tests was reported, and if a problem reporting mechanism was used to facilitate the correction of software or specification errors. Specifically, we asked whether:

- The project produced a test plan and test designs, and secured customer concurrence on them.

- The project produced test procedures and data cases and placed them in a controlled library to facilitate test repeatability.

- The test procedures included expected results in terms of pass/fail criteria.

- The objective of the tests was to ensure that the code correctly implemented the design.

- The project established a means to report test status and problems encountered in testing.

- The project established procedures to confirm compliance with the test plan and conduct.

- The project actually performed their test design and construction activities, and conducted their tests according to their test procedures.

3.6.4 Implementations By Projects Surveyed

Two of the projects (Project B and D) conducted controlled and formalized testing, involving an orderly progression through checkout, integration, functional, and acceptance testing. These projects used the Unit Development Folder as a vehicle for recording test procedures, data, pass/fail criteria, and test results. Both of these projects involved their customers in test planning and in the determination of test compliance. Software elements under test went through an orderly progression of controlled configurations, from "development" to "prototype" to "accepted"; project review and customer concurrence was employed in moving each element through the progression.

A third project (Project A) conducted somewhat less formalized checkout testing. For this project, the objective of determining that the code correctly implemented the design was accomplished; the design documentation included test cases and pass/fail criteria which were used to structure the checkout activity. This project employed the User Guide drafted during component specification activities as a prime vehicle for functional testing, thus assuring the requirements documented in the User Guide were in fact met in the developed system.

The other two projects conducted functional testing based upon system requirements, with varying degrees of formality. In one case (Project C) the requirements were stated in such a way as to make objective testing with specific test criteria difficult. In the other case, a specified set of benchmark tests which had been conducted on an earlier version of the system constituted the basis for functional testing; the results of re-running these benchmarks against the newly-developed system were conclusive and verifiable. Control of the configurations under test was informal. Acceptance testing for these two projects was conducted with a subset of the functional tests. Customer involvement was limited to the acceptance testing activity.

PRACTICE	PROJECT A	B	C	D	E
Acceptance Testing					
Requirements Identified		X		X	X
Test Plan		X		X	X
Review for Quality				X	
Readiness Review		X	X		
Acceptance Test Rehearsal	X	X		X	X
Control Listings and Milestone		X	X	X	X
Acceptance Report		X	X	X	X
Customer Buy-Off		X	X	X	X
Procedures Stated		X		X	X
Activity Performed		X		X	X
Functional Testing					
End Item Inspection		X			
End Item Hand Over	X	X	X	X	
Independent Test		X	X		X
Test Plans	X	X	X	X	X
Pass/Fail Criteria	X	X		X	X
Code Satisfies Requirements	X	X		X	X
Problem Reporting		X	X	X	
Control of Activity		X	X	X	
Procedures Stated		X	X	X	X
Activity Performed		X	X	X	X
Test Formalism					
Test Plan and Procedures		X		X	
Pass/Fail Criteria	X	X		X	
Code Implements Design	X	X		X	
Problem Reporting and Status		X	X	X	
Control of Activity		X		X	
Procedures Stated		X		X	
Activity Performed		X		X	

INDICATORS OF TESTING METHODOLOGY BY PROJECT

FIGURE 3-8

3.7 CONFIGURATION MANAGEMENT AND CHANGE CONTROL

The practices of software configuration management and change control (CM/CC) are becoming both better defined and more effectively applied. As discussed in Section 2.0, the traditional focus for CM/CC has been what the software developer saw as his primary product: computer code. Other products, particularly documentation, have either been ignored in the practice or have been assumed to be adequately covered by other control mechanisms (such as formal document release procedures).

The objective of traditional software CM/CC has simply been to prevent changes. Usually the stated objective has been to preclude "unauthorized" or "unexpected" changes, but the traditional method -- of "locking up" the code -- has most often had the effect of preventing any changes at all.

In practice, this simplistic method has not provided the desired result: a mechanism for orderly, cost-effective introduction of needed changes. Most commonly, during integration when the pressures to meet schedule are most severe, even the pretense of control is cast off, and what little visibility and control of changes may have existed previously is simply lost.

Previous discussions have identified a number of changes in the software development environment. Three of these changes have major implications for the practice of software CM/CC: the need for close control of costs, greater attention to the functional requirements of the software, and increased concern for the apparent quality of the software product.

In the realm of cost management, the emphasis is on making sure that the work performed (and the resources expended) is really required to satisfy contractual obligations. In particular, the Program Manager needs to be sure that any changes in the work plan -- and, hence, in the products of that work -- are instituted only after due consideration of responsibility. For example, changes to correct coding errors or document "typos" are clearly in-scope modifications, to be implemented at the developer's expense. (The Program Manager will usually have budgeted for this kind of work in his program plan.) However, addition of a new capability to the software, such as a new report format, is obviously out-of-scope; the resources (and schedule and budget) to do work of this nature must be added to the program plan, after the customer has concurred with the need and provided incremental funding.

While changes like these may be easy to categorize, most product changes will not be quite so obvious. A typical example is the situation where the customer finds an error in one of the algorithms he has supplied, and the developer could repair the problem with a simple change in the code. Most commonly, the developer will make the code change as an accommodation, treating it as in-scope just to save himself and the customer the paperwork needed to obtain what is probably a small amount of additional funding.

Only in retrospect will the developer and the customer realize that this simple change has propagated through several documents, training materials, and even test cases -- with a total cost far greater than that of just the code update. The Program Manager finds that effective cost control requires

effective product control. That is, there must be visibility, at the item level, of all of the products of software development and how they relate to providing the specified, required computing capability.

Discussion earlier in this section (in 3.1 and 3.2) centered on the procedures by which a Program Manager determines what items are planned to be produced, and which of these have been developed. With the structured development process that is becoming Modern Programming Practice, some of these items may be packaged and given to the customer at milestones (PDR, CDR, or PCA/FCA) prior to final delivery. A change found necessary in one of these items must be handled with special care, if only to avoid confusion with the same item delivered previously.

Having recognized the need for comprehensive cost and product control, the Program Manager must face a much more pervasive question; namely, how the need for a change will be identified. Many people have a more or less direct involvement with the software: the users, the designers, the programmers, even the computer operators. Any of them may encounter (or anticipate) a problem. If the problem discoverer cannot find a way to report the problem or, having done so, cannot learn what is being done about it, there is potential for real dissatisfaction, even though the software itself may completely meet the approved specifications.

It should be noted here that a "problem" need not necessarily be the result of a fault in the software. Much more often, the "problem" may be symptomatic of a need for additional capability, which the customer and developer did not recognize when requirements were originally defined. Or, the problem may be that the documentation implies existence of a capability (or usage) not intended. Or, as cited earlier, the users may discover that the algorithms they have specified are inappropriate for the application.

Recognizing the need to provide a mechanism by which problems can be reported, it becomes apparent that two additional mechanisms are required. One of these lets the person who has reported a problem review the change developed in response to it, to confirm that the reported problem has been resolved. The other mechanism provides a way of distributing the change to all who will be affected by it.

To meet these changing needs, three practices already common in the engineering world are being applied in the development of software. One of these is the practice of explicitly identifying a software configuration called, in BCS terminology, a baseline. A baseline specification cites a particular configuration of software items which provides a designated set of capabilities. Another practice is the use of problem reporting/resolution systems; to provide a focal point for reporting problems and obtaining reliable information about the status of a computing capability. The third practice is the institution of Change Control Boards, to authorize the development of changes to repair reported problems, and to approve distribution of software packages implementing proven changes. These new practices will be discussed further in the following paragraphs.

3.7.1 Baselining

The practice of baselining employs a configuration indexing scheme (discussed earlier in the context of PCA and related Modern Programming Practices for program management). Basically, a configuration index is simply a list of all of the task results (to be) produced during the Construction phase, including the end items (contract deliverables) into which these results are incorporated. This list serves the Program Manager's needs by providing a basis for task scheduling and resource planning; as task results are produced and signed off, the Program Manager uses this additional information as evidence of task completion, the key indicator in his determination of status, progress, and performance against the program plan.

A baseline extends the configuration index by specifically identifying 1) the set of capabilities implemented, 2) the constituent(s) which provide each capability, and 3) the implementation (i.e., configuration item) of each constituent. The capabilities list is extracted from -- and references -- the requirements definition approved at PDR: it also references the PDR-approved product specification, to indicate which software component (including documentation and other products of the development activity) provides the specified capability.

The constituents list expands on this by abstracting, from the designs and plans approved at CDR, the identifiers of the items to be produced by development tasks. This, of course, is the basic configuration index. To provide information for later use in the event a change becomes necessary, the constituents list also records the planned resource expenditures for each item.

The implementations list, finally, is the configuration index augmented with the information available at PCA. That is, it now lists the controlled repository in which each configuration item has been placed, and the name of the person responsible for its control. The list may also include such additional data as the actual expenditures incurred in development and the date the item was completed/accepted.

The totality of these is a baseline. Simply stated, a baseline is a mechanism for cross-referencing requirements, specifications, designs, plans, and products of the development effort. Given this basic mechanism, a relatively straightforward extension of the identification scheme suffices to accommodate changes.

BCS has adopted fairly standard terminology for the qualifying identifiers of a baseline. They are, in order of precedence:

- The system name, usually an acronym, which connotes a class of applications (such as structural analysis, or financial modelling, or computer performance measurement) the capability package is intended to support.

- The version identifier, which distinguishes systems having the same name and application, but significantly different sets of capabilities (for example, performance measurement of Honeywell computers, as distinct from that for Univac machines).

- The release designation, which identifies the set of capabilities/constituents/implementations currently approved for operational use, and distinguishes between this set and prior releases, or planned future releases.

- The revision code, which designates a particular set of changes to an approved release.

(In some situations, particularly when a software system is to be used in two or more computer facilities having significantly different configurations of equipment and support libraries/facilities, still another qualifier may be employed. The facility designation, interposed between release and revision qualifiers, identifies changes implemented to accommodate local conditions.)

In practice, a baseline is created for each approved combination of system, version, and release qualifiers. That is, the designated configuration items are copied into controlled storage, and all members stored are labelled to indicate the baseline to which they belong. A baseline is not created for each revision (or facility) qualifier, but a configuration index is set up by which a baseline can be created if the decision is made to approve the revision for formal release. This requires that item naming conventions allow having two or more implementations of a given item in controlled storage, distinguishing among such items by designating the revision to which each belongs.

There is as yet no "standard" practice for baselining software nor is there much likelihood of such a standard emerging until there has been substantial improvement in the tools used in this practice (see discussion of programming support library aids in Section 3.5). When we queried our projects for this study, we posed our questions on this topic expecting a substantial degree of pragmatism. Specifically, we looked only for evidence that:

- Written project procedures established the mechanisms, by project phase, for identifying the items produced.

- Written project procedures specified that controlled baselines were to be established, at least following the formal reviews of PDR, CDR, and PCA/FCA.

- Written project procedures defined responsibilities and mechanisms for updating and distributing approved baselines.

- The project deliberately verified that these procedures were implemented and followed.

3.7.2 Problem Reporting and Resolution

The primary impetus for implementing a problem reporting and resolution system is to help ensure user satisfaction. As we will see in the following discussion of Change Control Boards, however, such a system is a key element in being able to exercise effective control of project costs.

There are three essential features of a problem reporting and resolution system: 1) a focal point for reporting problems and obtaining status about resolution; 2) positive feedback on the action taken to resolve a problem; and 3) a mechanism for closing out a problem once resolution action has been completed.

Properly speaking, the Program Manager -- as the designated person having responsibility for the software development -- is always the focal point for reporting problems and obtaining status. In practice, the PM will usually delegate this function to some individual in his organization (or, if the software developers and users are geographically separated, at each using facility). This person is charged with receiving reports of "problems" (with supporting evidence), and assigns to each an identification code for use in tracing the problem through subsequent resolution action. This person has the additional responsibility of receiving information about the actions taken, and disseminating this as appropriate to interested enquirers.

The feedback mechanism will ordinarily have two elements. The first of these is to have the focal point contact the problem reporter, once disposition has been decided, to advise whether the problem was accepted for resolution and, if so, when correction can be expected. The second part of the mechanism operates if the problem was accepted for correction. Once the change that resolves the problem has been developed, a revision (of the system) containing the change is submitted to the reporter so he can verify whether the problem originally reported has been corrected.

Close out of a problem can be accomplished in either of two ways. After the problem report is received and analyzed, the development organization will ordinarily write a change proposal, identifying the configuration items that must be modified to effect resolution, describing the change to be made in each item, and estimating the probable cost of doing the work. The Change Control Board (see next discussion) will either authorize the work, setting a date for completion, or request revision of the proposal, or reject the proposal. The latter action in effect closes the issue. The focal point then contacts the problem reporter to advise that the problem will not be resolved and cites the reasons given by the Board for taking this action.

The other manner of close out occurs after the problem reporter has verified the resolving change. At some later time, the change will be incorporated in an approved release of the system, placing the correction into operational status. The problem reporter is advised of plans for this, and the problem is officially closed when it occurs. (It should be noted that, even though a change to correct a problem has been developed and verified, the Change Control Board retains the option of not approving it for release. The problem

reporter would be advised of this kind of closure action if it is taken, and the reasons for it.)

In structuring our project survey, we did not expect to find significant commonality of software CM/CC practices. In the case of problem reporting and resolution, we anticipated only that the projects would have recognized a need for some formality in the process. Rather than postulate that some emerging "standard" would be discernible, we looked only for evidence that some kind of problem record-keeping had been instituted. Specifically, we asked whether:

- Written project procedures established mechanisms for formally reporting problems (including desired improvements) in the software package, for identifying, processing, and tracking problem reports, and for disseminating/obtaining problem status information.

- These procedures were implemented and followed (evidenced, in part, by a file of completed/in-progress problem reports).

3.7.3 Change Control Board (CCB)

Throughout the preceding discussions of Modern Programming Practices, there has been repeated emphasis on the interaction between the developer and the customer of a computing capability, to define the requirements, to specify the product, and to concur in the work plan. The central issue in this section is that a change in any one of these must ordinarily be reflected by corresponding changes in the other two. It is this fact that traditionally has made both the developer and customer strongly resist any kind of change at all.

This resistance to change has usually been predicated on the difficulty of ensuring that all implications of a change have been adequately considered and accommodated. With the Modern Programming Practices of comprehensive task planning and item identification, tied to requirements by means of baselining, there is much better visibility of the development process. This visibility now makes it feasible to implement the third key mechanism of software CM/CC, the Change Control Board.

By BCS policy, a Change Control Board is constituted of persons 1) knowledgable of the customer's mission and the intended use of the computing system and its software, 2) representing the interests of the user, developer, operator (i.e., facility), and sponsor of the software, and 3) authorized to make commitments on behalf of the organizations they represent. The purpose of this policy is to ensure that due consideration is given to all aspects of a proposed change and, if the change is authorized, that all actions needed to implement it will be carried out.

In practice, a software Change Control Board is chaired either by the Program Manager or by a representative of the customer. The Board meets periodically, or as required to review and act on proposed changes. As a rule, the chairman will specify the required participation for each meeting as a function of the implications of the changes to be acted on.

The Change Control Board has two primary responsibilities: to act on change proposals, and to approve distribution of baselined systems. In the former responsibility, it reviews the problems reported and decides the proper disposition of the changes proposed for their resolution.

The problem review must consider a number of factors. The Board must determine whether the problem, if left unresolved, would critically impact the customer's ability to carry out his mission. If so, the Board must determine the priority of the problem relative to any others in this category. (For example, Problem A may be crucial to an initial operational capability, while Problem B may not become significant until a later time.) Also, they must consider the possible side effects of implementing a solution. (The users may already be familiar with the problem and have devised appropriate workarounds; correcting the problem may require substantial re-training, with the result that the solution could cost more than it is worth.) And there is the matter of available funding -- whether the change is in-scope or out-of-scope -- and whether (or when) the budget can pay for the needed work.

If a proposed change is deemed necessary (that is, the problem is significant and of sufficiently high priority to warrant correction), the Board will authorize development to proceed. Specifically, they will assign action items to the various organizations that must participate in developing and testing the change, set a schedule for accomplishing the work, and specify funding authority for the effort.

It should be noted here that the Board must also consider possible interactions of changes. That is, two (or more) reported problems may require for their resolution that the same configuration item be changed, perhaps in different ways. Looking ahead to the time that corrections to both problems will be eligible for release, the Board will have to authorize a third change, one that combines the effects of the other two. By this means the Board retains its option to release one change or the other, or both.

After the change has been developed, tested, and submitted to the problem reporter(s) for verification, the Board can act on its second responsibility: approving the distribution of a new baseline. Ordinarily, this action will be taken only at pre-defined intervals (each four months, say) or at times in a customer's mission when release of a system update is appropriate. At these times there will usually be a backlog of verified changes. The Board must determine which of these it is appropriate to release to operational use (considering possible impact on established procedures and the need for re-training).

In practice, the Program Manager will submit, for consideration by the Board, the specifications of one or more baselines he feels are suitable combinations of capabilities and changes for release. Where more than one specification is submitted, the Program Manager will provide selection rationale for each. The Board will then act to approve release (of one), setting a schedule for implementation and buy-off and, usually, establishing a fall-back plan should the new package somehow prove inoperable. With the actions assigned by the Board, it is the Program Manager's responsibility to get the updated system into operational use.

Traditionally, the implementation of changes has been the foundering point for software project cost control. The Modern Programming Practice of involving a Change Control Board as the focus for deciding on changes provides an opportunity to significantly improve the traditional situation. For the projects we studied, then, we were particularly interested in whether and how they had responded to this opportunity. We looked for evidence that:

- A CCB was organized with appropriate management representation, including the customer, who possessed decision and budgetary authority to effect control over baseline changes.

- The CCB was responsible for resolution of reported problems, assessment of proposed changes, prioritization and control of change implementation, and authorization of baseline updates and distribution.

- The project verified that the CCB was operational and discharged its responsibility.

3.7.4 Implementations By Projects Surveyed

Three of the projects surveyed (Projects A, B, and D) instituted baseline control for requirements, designs, and the implemented system. Two of these projects attached particular importance to their establishment of baselines. In one case (Project A), the Program Manager pointed out that the requirements baseline documented in the User Guide became the prime basis for functional testing of the implemented system. In the other case (Project B), the Program Manager felt that establishment of requirements and design baselines materially contributed to the control of changes and their impact during the Design and Construction phases.

The other two projects (Project C and E) instituted baseline control shortly before PCA/FCA, in accordance with traditional practice.

Two of the projects surveyed (Projects B and D) instituted a problem reporting system encompassing the entire development cycle. Two other projects (Projects A and C) utilized a problem reporting mechanism during only a portion of the development. One of these (Project A) tracked problems and changes only during the Design phase; the other (Project C) instituted formal problem reporting and resolution during the testing activities. The project which instituted a problem reporting system at the onset of testing did so in response to critical problems with the developed code made evident by functional testing; this project felt that the institution of this practice was critical to the success of their testing efforts and to the ultimate delivery of the developed system.

Two of the projects surveyed (Projects B and D) had a formally constituted Change Control Board (CCB) with appropriate project and customer representation. The CCB was active from the establishment of the requirements baseline throughout the remainder of the project, and was considered a key element in maintaining customer visibility of the development as it proceeded. The Program Manager in one case (Project B) expressed the opinion that the assessment and prioritization of problems and changes which the CCB accomplished allowed the development activity to be more responsive to changing customer needs.

PRACTICE	PROJECT A	B	C	D	E
Baselining					
End Items Identified	X	X	X	X	X
Baselines Exist	X	X	X	X	X
Update/Distribution Mechanism		X	X	X	
Procedures Stated	X	X		X	
Activity Performed	X	X		X	
Problem Reporting and Resolution					
Reporting Mechanism	X	X	X	X	
Tracking of Reports		X	X	X	
Distributing Reports		X	X	X	
File of Reports	X	X	X	X	
Procedures Stated	X	X	X	X	
Activity Performed	X	X	X	X	
Change Control Board					
Project/Customer Representation		X		X	
Authority of Board		X		X	
Assess Changes		X		X	
Prioritize Changes		X		X	
Authorize Updates				X	
Procedures Stated		X		X	
Activity Performed		X		X	

INDICATORS OF CONFIGURATION MANAGEMENT AND CHANGE CONTROL BY PROJECT

FIGURE 3-9

4.0 DATA ANALYSIS

This section contains a description of the method we used to perform our analysis, discussions of the effects we were able to discern from the data we gathered, and our interpretations concerning which specific Modern Programming Practices were responsible for these effects. Conclusions concerning the effects of some of the observed MPP are presented in the final portion of this section.

To meet the objective of this study, we postulated two basic hypotheses (see Section 4.1), collected data to test these hypotheses, performed the tests and analyzed the results. We found that MPP in two categories (Program Management and Testing Methodology) appear to have significant cost benefits for software development. A third group of MPP (Configuration Management and Change Control) may have a "payoff" in the later stages of development, especially for large projects. While our data do not directly substantiate this, a fourth set of MPP (Design Methodology) were universally viewed by project participants as significantly beneficial.

4.1 ANALYSIS METHOD

Our study was to be based on a very small sampling of experience, and the data we would gather from the projects was expected to be sparse and somewhat tainted. We devised an approach which would rely, as much as possible, on tangible evidence - project records and documented products of development activities. We were particularly concerned with acquiring a complete picture of project costs (whether or not they were formally part of the project's financial record). We wished to assure ourselves that when a project claimed to use a practice, that practice was, in fact, employed consistently. Our analysis method, while informal, has allowed us to reach some conclusions which we believe are valid.

4.1.1 Hypotheses Formulation

The objective of this study was to discern how Modern Programming Practices affect the costs of software development at BCS. Given this objective, we postulated that the application of MPP could affect software development costs in several ways. Projects employing MPP might experience either cost reductions or cost increases. Alternatively, even if total software development costs did not significantly change, the application of MPP might cause costs to be redistributed, i.e. some activities might become more costly, and some might become less costly. Considering these possible outcomes, we formulated two independent hypotheses for testing in our study.

4.1.2 Hypothesis I

Our first hypothesis was that the use of MPP significantly changes the total costs of software development. More formally, Hypothesis I can be stated as follows:

Hypothesis I: There is a significant quantitative effect on the total costs of software development when (some subset of) Modern Programming Practices are employed. This effect is measurable as the difference between forecasted and actual costs, where traditional forecasting techniques have proven accurate to within ± 15% of expenditures (with a 90% confidence level).

We believed that our data analysis had to show that this hypothesis was valid if we were to be able to draw any further conclusions about the effects of specific MPP. In other words, to analyze the cost effects and attempt to attribute them to MPP employed on the projects, the effects first had to be discernible (i.e. greater than ± 15%).

4.1.3 Hypothesis II

Our second hypothesis was that the use of MPP causes a re-distribution of resource expenditures during software development. More formally, Hypothesis II can be stated as follows:

Hypothesis II: In a project which employs (some subset of) Modern Programming Practices, software development costs assume a different distribution than that which is traditionally experienced. In particular, resource expenditures for the activities of definition and design increase, while expenditures for construction activities (especially testing) decrease.

Hypothesis II was formulated primarily as a "fall-back" position, in the event that we were not able to prove Hypothesis I valid. Even if no discernible change in total software development costs was found, we believed we might see that resources were apportioned differently. This would imply that MPP usage has effects on other aspects of concern to both developer and customer - product quality, schedule risk, and/or costs of operation and maintenance.

4.1.4 Questionnaire Development

We developed a questionnaire as a vehicle for gathering data from the projects selected for study. This questionnaire focused on three pertinent issues: the specific Modern Programming Practices employed by a project, the actual expenditures incurred by the project in developing their software system, and the product and project characteristics which would allow us to forecast development costs. A specimen questionnaire is provided in Appendix C.

4.1.4.1 MPP Data

Our intent in gathering data concerning MPP implementation was, first, to identify which practices were employed by a project, and then, for those specific practices, to gather detailed data concerning tangible evidence and formality of implementation. The first section of our questionnaire, therefore, simply lists the practices. Later sections of the questionnaire contain detailed questions concerning each practice. These questions were based on the indicators of practice implementation described in Section 3. The responses to these detailed questions were the basic MPP data we used in our analysis of practice effects.

4.1.4.2 Expenditure Data

Our questionnaire also solicited information from the project's financial records concerning both labor and computer resource expenditures. Realizing that two of our five projects were on-going, we requested actual expenditures for completed activities and estimated expenditures for current or planned activities. We designed our questions to partition reported expenditures attributable to at least "prior to coding" and "coding through delivery" activities, so we could use the responses to test either of our hypotheses.

4.1.4.3 Estimating Data

In order to forecast development costs for each project, we included questions designed to identify appropriate estimating parameters. The estimating technique used to forecast costs is described in Section 4.2. The estimating data we required included: type of software and the end product size (in numbers of source statements), and adjustment factors such as experience level of personnel, use of a higher-order language, and availability of debugging tools. The forecasts we developed with this data were compared with the actual expenditure data to test our hypotheses.

4.1.5 Interview

We conducted a series of interviews with each project to gather the data for our study. These interviews were conducted in person or via conference call, and involved the project's Program Manager and, in most cases, key project personnel whom the PM felt could supply additional insight. During the interviews, we stepped through the questionnaire, asking each question and recording the answers. The direct interview permitted us to obtain more complete answers to our questions; where a project employed a "local jargon" to describe a particular practice, we were able to ask amplifying questions which allowed us to relate responses more directly to our objective. Since we assured them that project identity would be kept confidential, we were also able to obtain subjective views from the project participants concerning the practices they employed. In addition, the interviews permitted us to discover unique situations such as unanticipated personnel turnover which we could consider in our data analysis.

4.1.6 Analysis

In order to analyze the data we gathered for this study, our intent was first to test Hypothesis I to determine if a significant cost change was evident. If such a change could be discerned, we could then study the indicators of practice implementation and the results of our interviews to identify cause/effect relationships. We intended to quantify these relationships, where possible. If the test of Hypothesis I failed, we would then resort to testing Hypothesis II and analyzing the effects we discerned.

4.1.6.1 Testing of Hypothesis I

The estimating data we gathered in our project interviews permitted us to develop, for each project, forecasts of total labor expenditures (see Appendix

D). These forecasts were then to be compared with the actual expenditure data our MPP using projects supplied. If a significant and consistent cost effect could be identified, then we could proceed to interpret our MPP data.

4.1.6.2 Interpreting MPP Data

Given the effects which we identified in testing Hypothesis I, we could then compare the MPP data gathered in our project interviews to determine the impact of specific practices. We intended to isolate those practices (or practice indicators) unique to the projects experiencing the most significant changes. Realizing the interdependence of the practices under study, we anticipated that we would only be able to isolate groups of practices. Further, we expected that the unique project characteristics noted in our interviews would influence the degree to which implemented practices could be considered direct causes of the effects under examination.

4.1.6.3 Quantifying Relationships

Having identified (groups of) practices which appeared to cause the cost changes, we intended to examine the magnitude of the changes and quantify the effects of the practices. We hoped to be able to identify specific practices whose "leverage" on development costs was significant enough that we could draw conclusions about their applicability.

4.1.6.4 Testing of Hypothesis II

If we were not able to discern a change in total development costs in testing Hypothesis I, then we intended to examine our cost data for expenditure shifts. In this case, our analysis and interpretation would consider whether implementation of MPP caused a discernible increase in resource expenditures for certain activities, and a corresponding decrease in expenditures for others.

4.2 HYPOTHESIS I VALIDATION

Having determined that some significant changes have occurred in the role the software developer assumes in supporting his customer, and in the specific practices the software developer uses to do his work, we anticipated that the total cost associated with software development would be different than what we have historically encountered.

4.2.1 Estimating Technique

The existence of historical data on software development costs makes it possible to forecast with some accuracy costs for development undertaken in the traditional mode. These cost forecasts usually take into account such basic parameters as type of software application and number of executable statements in the end product, and are refined by adjustment factors which account for unique characteristics (e.g., the software is simply to be converted from one computer configuration to another, or the product

is to be coded in a higher order language, or on-line facilities for coding, data entry, and testing are available).

BCS has developed an algorithm for estimating software development expenditures; this algorithm is commonly used by us as a "reasonableness check" of our software development cost proposals.

The estimating factors included in the algorithm which BCS employs (see Appendix B) are drawn from several sources: BCS' own historical experience, the project cost histories which have appeared in various industry publications, and published Air Force studies. These factors permit the estimator of a proposed development to quantify the cost effects of commonly-employed productivity improvement methods, and therefore produce a reasonably accurate cost estimate for a specific project utilizing one or more of these methods. We have found that the algorithm we use produces cost forecasts within ± 15% of the actual total costs experienced on past BCS projects, with a 90% confidence level.

This algorithm, like others in common use within our industry, presumes the traditional role of the software developer, and the traditional practices with which the developer is most familiar. For this reason, we felt that this algorithm would be a reliable vehicle for testing Hypothesis I. In our data gathering activities, we deliberately solicited information from the projects selected for study which would allow us to forecast (traditional) development costs via this algorithm.

It should be noted that we opted for this approach rather than relying directly on the Program Manager's own estimate of his project costs. We did this to avoid the possibility that the PM's development estimate might have built into it his own intuitive bias on what the cost effects of Modern Programming Practices might be.

Since the estimating guidelines are only accurate to within ± 15%, we felt that significant impact, as postulated in Hypothesis I, would have to be well above this "accuracy band". In other words, we did not anticipate that we would be able to distinguish the impact of Modern Programming Practices, if that impact was not so large as to cause a difference in forecasted and actual development costs greater than ± 15%.

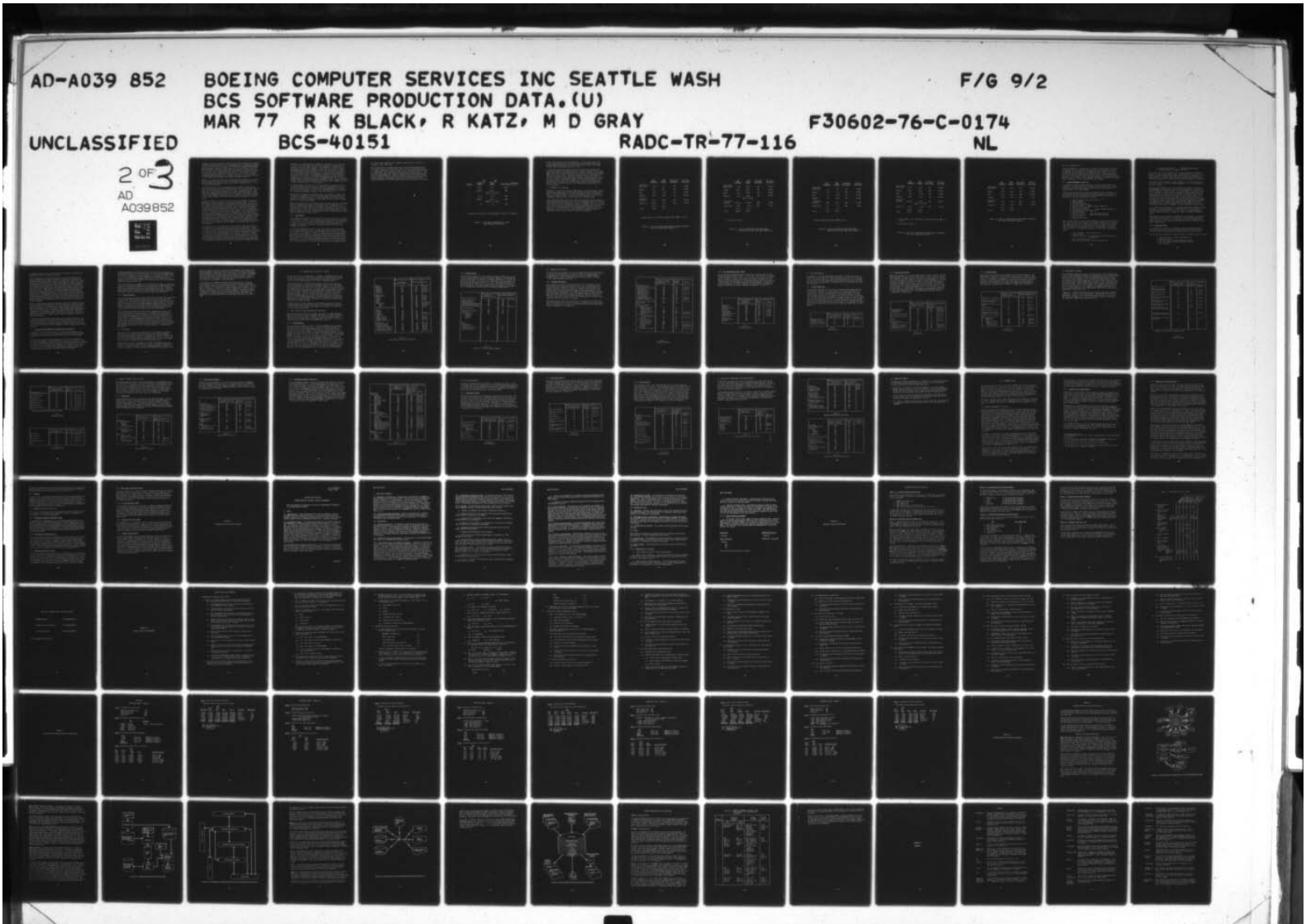
Hypothesis I was formulated in such a way as to avoid pre-judging the outcome of our data analysis. That is, Hypothesis I makes no statement of whether the cost effect of MPP implementation would be beneficial or detrimental, but only that the impact would be significant enough to be detectable.

4.2.2 Anticipated Outcomes

Having reviewed the descriptions of BCS' Modern Programming Practices in Section 3 of this report, one might expect that the effect of MPP implementation on total software development costs would in fact be detrimental, i.e. the implementation of MPP would significantly increase costs. This expectation is supported by points made in Section 3. For example, the fact that the software developer is now producing several documents aimed

AD-A039 852 BOEING COMPUTER SERVICES INC SEATTLE WASH F/G 9/2
BCS SOFTWARE PRODUCTION DATA.(U)
MAR 77 R K BLACK, R KATZ, M D GRAY F30602-76-C-0174
UNCLASSIFIED BCS-40151 RADC-TR-77-116 NL

2 OF 3
AD
A039852



at specific, distinct audiences (instead of the omnibus Maintenance Document), and that these documents are now being produced early in software development (rather than after delivery as part of the Operation and Maintenance phase), should add costs (which were traditionally part of O & M costs) to the software development lifecycle. Another additive cost could be postulated from the fact that, simply because MPP are new and unfamiliar, projects which are now employing the practices have to bear costs of training their personnel in the implementing procedures, techniques, tools, etc. (This cost would presumably become negligible once the practices come into reasonably consistent and common usage throughout the computer software industry.)

However, in undertaking this study, we expected that the results of our data analysis would show that total costs for software development actually decrease when MPP are employed. Several observations supporting this expectation are discussed below.

As discussed in Section 3, the new emphasis on rigorous top down design has an influence on the manner in which software testing can proceed. Because design and code production is performed top down, the programmer no longer must resort to building "throw away" test driver programs to check out portions of a component. Production of test drivers can, traditionally, become a significant consumer of resources, and can complicate the testing process itself, since the code in the driver must also be determined to be free of errors.

On several occasions in Section 3, we noted that the customer is becoming increasingly involved during the Definition and Design phases. That involvement implies that the customer's requirements are being more deliberately and formally defined in detail, and that they are documented (and concurred with by him). Requirements are, therefore, less likely to be overlooked or forgotten as design and implementation proceed. In addition, the customer is more involved in the design verification process. This involvement permits a realistic evaluation of the software design against stated requirements, which can uncover omissions or errors in design and in requirements before the programmer has propagated those errors into elements of code. We expected that the result of this increased (and earlier) customer involvement in the development process should be that fewer fundamental (and therefore costly) errors go undetected until functional testing or formal demonstration. In other words, we believed that the programmer is now re-developing and adding less code during the Construction and Demonstration phases than he has traditionally.

Another observation can be made about the costs which traditional software development projects experience. An inspection of the adjustment factors in the BCS software estimating guidelines (Appendix B) points out a symptom of what has historically been a significant problem. When projects are "large", i.e. when more than ten programmers are involved in a software development, the effect on total resource expenditures is adverse, particularly during testing activities. We attributed this to a lack of communication and coordination between programmers, each of whom is responsible for a single functional component. Traditional practice has been to isolate these components and allow each programmer as much latitude as possible in designing, coding,

checking out, and documenting his component. The impact of this isolation is then felt during integration, when the programmers' products must be altered so that they "work" together; the propagation of changes during integration testing usually becomes a significant cost factor. The Modern Programming Practices which comprise the new design methodology require early and complete design documentation, written according to representation formats whose content and meaning are established and understood as project standards. Further, a programmer's designs are being subjected to independent verification via structured walkthroughs, and are available to other project personnel in the Unit Development Folder. We felt that the communication and coordination between programmers which is possible because of these design and documentation practices should serve to dampen the (traditional) cost effects of a "large" project.

Our observations also pointed out that the communication problem inherent in "large" projects should be further lessened by the emphasis on written, detailed task descriptions, produced by the Program Manager, since these task descriptions explicitly identify the individual(s) who are responsible for reviewing a programmer's designs, code, test results, etc.

The practice of preparing a draft User Guide in the form of a Product Specification for customer review at Preliminary Design Review should serve intra-project communication as well; each programmer can obtain an overall concept of the external behavior of the software system, and how his component "fits" into that environment.

Finally, we observed that the new practices of software configuration management and change control, and the planning and control responsibilities inherent in the Program Manager's role, have as their objective the control (or reduction) of costs. In practice, however, the amount of detailed information required to carry out these practices places a significant (clerical) resource burden on a project, which we expected would balance the cost control benefits these practices might provide.

4.2.3 Test Results

The information we gathered in our project interviews allowed us to forecast, for each project, the total labor expenditure the project could be expected to incur had it used only traditional programming practices. (The derivations of these forecasts are presented in Appendix D.) We then compared these forecasts of traditional costs with the actual expenditures incurred by our MPP-using projects.

It must be acknowledged that labor is only one of the resources expended in developing computer software. A second significant contributor to costs is, of course, the computer time used in compiling and testing the software. The problem is that "computer time" is not an easily quantified parameter, particularly with today's multiprogrammed, multiprocessor hardware systems. While the estimating guidelines used in this study to forecast labor costs also provide rules for predicting computer usage, we elected to not include this factor in our study. This decision was the result of our determining, early in the planning for the study, that there existed no common rationale

for trading labor expenditures and computer usage that was not heavily influenced by local conditions.

When we compared our forecasted labor costs with the actual expenditures the projects reported, we found that three of the five projects (Projects A, B, and D) experienced significant total cost reductions. Figure 4-1 shows this comparison. (Note that two of these projects - Projects B and D - are on-going; their actual labor expenditures are shown as expenditures to-date *plus the Program Manager's* estimate to complete.) While the remaining two projects (Projects C and E) also incurred lower than predicted costs, these differences cannot be considered significant since they are within the ± 15% tolerance of the estimating guidelines we used to create the forecasts.

Project	Forecasts (MM) (1)	Actuals (MM) (2)	% Difference $\left[\frac{(2)-(1)}{(1)}\right]$
A	358.2	71.0	-80%
B	2,288.5	991.6 [617.3 + 374.3] *	-57%
C	51.5	43.8	-15%
D	3,298.7	514.8 [439.6 + 75.2] *	-84%
E	7.9	7.3	- 8%

* Expenditures to-date plus Program Manager's estimate to complete.

FIGURE 4-1 TRADITIONALLY FORECASTED VS. ACTUAL LABOR COSTS FOR EACH PROJECT

We can, therefore, conclude that Hypothesis I is valid for three of the five projects. Further, we can proceed to examine which MPP these three projects used which Projects C and E did not, and attribute to those MPP the cost benefits achieved (see Section 4.4).

An additional observation should be made here concerning the magnitude of the cost benefits enjoyed by Projects A, B, and D. Projects A and D both experienced benefits in the 80% range, while Project B benefitted only about 60%. As will be discussed in Section 4.4, Projects D and B were both "large" projects, and used virtually identical practices. One of the factors we believe contributed to the lesser benefit Project B enjoyed was that this particular project experienced a rather severe and abrupt personnel turnover just after their Critical Design Review, i.e. just as the design activity was completed. This implied to us that Project B had to bear additional costs in activities after CDR to bring in new personnel and acquaint them with the project.

4.3 HYPOTHESIS II VALIDATION

Hypothesis II stated that the use of MPP causes a re-distribution of resource expenditures amongst the activities performed during software development. While it was not necessary to test this hypothesis, since a validation of Hypothesis I was sufficient to satisfy our study objectives, we did perform a somewhat cursory analysis of the distribution of project costs.

These comparisons were based on the by-activity forecasts we developed with the software estimating guidelines. We should acknowledge that our estimating guidelines have been validated only for total development costs, and not for costs by activity within development. However, it appears that larger portions of the total resources expended are being consumed during definition and design activities, and smaller resource expenditures are occurring in integration and system testing activities. Figures 4-2 through 4-6 show the comparison of activity cost distributions for each project.

	(MM) Forecasts	(MM) Actuals	(Forecasts) % of Total	(Actuals) % of Total
Requirements Definition	31.9	6.0	9%	8.45%
Design	43.1	14.0	12%	19.72%
Code	23.3	7.5*	7%	10.56%
Checkout	25.9	18.7*	7%	26.41%
Integration and Test	127.7	18.8*	36%	26.41%
System Test	106.4	6.0	30%	8.45%
Totals	358.2	71.0		

* Code, checkout, and integration apportioned (45MM) via 2:5:5.

FIGURE 4-2 ACTIVITY DISTRIBUTIONS FOR TRADITIONALLY FORECASTED AND ACTUAL LABOR COSTS (PROJECT A)

	(MM) Forecasts	(MM) Actuals	(Forecasts) % of Total	(Actuals) % of Total
Requirements Definition	94.8	185.6	4%	18.72%
Design	111.7	172.4	5%	17.38%
Code	58.4	159.6	3%	16.09%
Checkout	60.9	199.6 [99.7 + 99.9*]	3%	20.12%
Integration and Test	1,353.6	178.0*	59%	17.95%
System Test	609.1	96.5*	27%	9.73%
Totals	2,288.5	991.7		

* Projected or planned

FIGURE 4-3 ACTIVITY DISTRIBUTIONS FOR TRADITIONALLY FORECASTED AND ACTUAL LABOR COSTS (PROJECT B)

	(MM) Forecasts	(MM) Actuals	(Forecasts) % of Total	(Actuals) % of Total
Requirements Definition	3.1	0.5	6%	1.14%
Design	5.7	8.3	11%	18.95%
Code	3.7	6.0*	7%	13.70%
Checkout	4.5	14.0*	9%	31.96%
Integration and Test	22.4	14.2	43%	32.42%
System Test	12.1	0.8	23%	1.83%
Totals	51.5	43.8		

* Code, checkout apportioned (20MM) via 2:5.

FIGURE 4-4 ACTIVITY DISTRIBUTIONS FOR TRADITIONALLY FORECASTED AND ACTUAL LABOR COSTS (PROJECT C)

	(MM) Forecasts	(MM) Actuals	(Forecasts) % of Total	(Actuals) % of Total
Requirements Definition	507.3	114.4	15%	22.22%
Design	430.4	125.2	13%	24.32%
Code	78.3	44.6*	2%	8.66%
Checkout	108.7	111.5*	3%	21.67%
Integration and Test	1,449.3	111.6* [43.9 + 67.7**]	44%	21.67%
System Test	724.7	7.5**	22%	1.46%
Totals	3,298.7	514.8		

* Code, checkout, and integration test apportioned (267.7MM) via 2:5:5.

** Projected or planned.

FIGURE 4-5 ACTIVITY DISTRIBUTIONS FOR TRADITIONALLY FORECASTED AND ACTUAL LABOR COSTS (PROJECT D)

	(MM) Forecasts	(MM) Actuals	(Forecasts) % of Total	(Actuals) % of Total
Requirements Definition	0.05	1.25	1%	17.24%
Design	0.29	2.25	4%	31.03%
Code	0.73	0.75	9%	10.35%
Checkout	1.90	0.25	24%	3.45%
Integration and Test	1.52	0.50	19%	6.90%
System Test	3.43	2.25	43%	31.03%
Totals	7.92	7.25		

FIGURE 4-6 ACTIVITY DISTRIBUTIONS FOR TRADITIONALLY FORECASTED AND ACTUAL LABOR COSTS (PROJECT E)

4.4 DATA INTERPRETATION

In testing Hypothesis I, we were able to discern a significant cost benefit for three of the five projects we surveyed. Our next step was to interpret the MPP data and the interview results we obtained in order to attribute these benefits to the specific set of MPP employed by those three projects. We first eliminated the MPP used by Projects C and E from consideration as cost benefit contributors, since these two projects did not enjoy significant benefits from their MPP. We then proceeded to examine the remaining practices used by Projects A, B, and D, and quantify their impact.

4.4.1 Program Management Practices

In comparing the practices and indicators of practices which Projects A, B, and D all evidenced and Projects C and E did not, we were able to reach some conclusions about those practices which directly contribute to the effectiveness of the Program Manager.

Examining the figures detailing practice implementations in Section 3, we first eliminated the practices used by Projects C and E as cost benefit contributors. Only the following were considered possible contributors to the benefits realized by Projects A, B, and D:

- Task Assignments (under PM Authority)
- Formal Reviews
- Document Pertinence
- Unit Development Folders
- Construction Plan } (under Design Completion)
- Review Prior to Coding }
- Interface Conventions } (under Coding Conventions)
- Code Organization and Comments }
- Code Verification
- Review For Quality (under Acceptance Testing)
- End Item Inspection (under Functional Testing)
- Test Formalism
- Change Control Board

Some of these practices were not used by Project A. Realizing that Project A was a "small" project, and Projects B and D were "large", we could discern that some practices could be considered beneficial regardless of project size, and some might only be beneficial for larger projects. Eliminating those practices not implemented by Project A, we were able to conclude that the following practices were beneficial for both small and large projects:

- Task Assignments (under PM Authority)
- Formal Reviews
- Document Pertinence (excepting Audience Review and Control of Changes)
- Unit Development Folders
- Review Prior to Coding (under Design Completion)

- o Interface Conventions } (under Coding Conventions)
- o Code Organization and Comments } (under Coding Conventions)

We felt that the last two items on this list were essentially "mandatory" practices. That is, even though Project A (and Projects C and E) did not formally establish standards for coding conventions, such standards in fact become mandated to satisfy computing facility requirements. We did not, therefore, consider these items to be significant cost benefit contributors.

The remaining practices have a common "thread:" all of them contribute to the effectiveness with which the Program Manager fulfills his responsibilities on a project.

Written task assignments are a basic communication vehicle, allowing the Program Manager to specifically identify what the assignee is required to do and the manner in which the work is to be performed. The definition and early production of audience-specific documents permits both customer and project personnel to visualize the end product, and understand how their responsibilities and contributions relate to that product. Formal reviews (with customer participation) and reviews of design consistency prior to coding tend to structure development activities. They provide both customer and project personnel visibility of what has been accomplished, the tangible evidence of that accomplishment, and what tasks remain to be done. Unit Development Folders provide the data which a Program Manager needs for effective planning and for assessment of status, progress, and performance.

All of these practices contribute to improved communication between customer, programmer, and Program Manager. It seems apparent from the magnitude of the cost benefits enjoyed by Projects A, B, and D that these avenues of improved communication have a substantial impact on software development costs.

Since Projects B and D employed some MPP not used in Project A, we examined these practices in more detail, to ascertain if certain MPP could impact costs for "large" projects. The testing and configuration management and change control practices employed by Projects B and D are discussed in the next two subsections.

4.4.2 Testing Practices

In examining the practices and indicators of practices which Projects B and D evidenced, we were able to reach some conclusions about the manner in which software testing methods can impact costs for large projects.

The practices and indicators of practices unique to Projects B and D included:

- o Construction Plan (under Design Completion)
- o Code Verification
- o Review For Quality (under Acceptance Testing)
- o End Item Inspection (under Functional Testing)
- o Test Formalism

The common "thread" evident in these practices is objective and detailed planning and conduct of software testing.

Detailed planning of the construction activities to be undertaken after design is completed includes the planning of activities to produce test procedures and test data cases, as well as scheduling of test runs in coordination with the completion of coding tasks. Code verification, through the mechanism of peer reviews, minimizes the number of errors that must be detected during checkout, and can also serve to ensure the adequacy of tests conducted during checkout, thus reducing the number of errors to be detected in later integration and functional testing. The reviews of end items for quality, completeness, and standards adherence which occur at the onset of acceptance and functional testing ensure that the software product and its related materials are in fact ready to be subjected to the planned tests. The formalism of test planning ensures that all tests are necessary and sufficient, and that their results will be conclusive. Test formalism also includes placing test materials in a controlled library to permit cost-effective reconstruction of test conditions for evaluation of software changes.

While our data does not show that these practices are of significant value to small projects (such as Project A), these practices do seem to promote more effective use of resources in the latter stages of the construction phase. An examination of the software estimating guidelines (Appendix B) reveals that, particularly for projects involving more than ten programmers, testing activities consume a significant amount of the total resources expended in a software development. Improved effectiveness in the performance of these activities should therefore cause a discernable lowering of total project costs.

It should be noted here that these conclusions concerning effects of test formalism are somewhat speculative, since neither Project B or D had actually progressed well into integration and functional testing at the time they were interviewed. Our analysis and conclusions are based on those project's estimates of what they would expend on testing rather than on actuals incurred.

4.4.3 Configuration Management and Change Control Practices

In examining the practices and practice indicators which Projects B and D evidenced, we were also able to reach some conclusions about the impact of Configuration Management and Change Control practices on development costs.

Only Projects B and D evidenced implementation of all three elements of Configuration Management and Change Control discussed in Section 3.7. Project A maintained baseline control and instituted problem reporting techniques only during the Definition and Design phases. Only Projects B and D had a formally constituted Change Control Board; as is evident in Section 3.7, this Board is vital in maintaining communication between developer and customer concerning the manner in which a system is evolving.

The Program Managers for Projects B and D, and others we interviewed, expressed the belief that these practices were in fact improving communication and visibility amongst all the participants in a software development. They felt that the Problem Reporting and Change Control Board practices, in particular, were causing more common recognition of interrelationships and responsibilities of each participant, and that more pertinent decision-making regarding changes was occurring.

Since the introduction of changes is often the primary reason for cost overruns near the end of development and in the Operation and Maintenance (O & M) portion of the software lifecycle, the practices used to control changes should show measurable effects in both development and O & M. It should be noted, however, that since Projects B and D are not yet complete, we cannot confidently state that actual practice will support this conclusion.

4.4.4 Design Practices

All of the projects we surveyed used the practice of top down design described in Section 3.3. Since Projects C and E did not achieve significant cost benefits, this practice was eliminated early in our data interpretation as a contributor to the benefits the other projects enjoyed. However, during our project interviews, all of the Program Managers expressed positive feelings about this method. Even where design verification was not rigorously practiced, the PMs stated that the combination of top down design and design verification contributed significantly to project effectiveness.

These subjective evaluations are not directly supported by our data, but we feel they should be given credence if only because the acceptance of these practices was so universal. Probably the improved customer/project and intra-project communication cited by our project participants has its primary benefit in establishing a better basis for formal testing. In particular, two of the Program Managers interviewed felt that testing activities proceeded much more smoothly, and fewer errors were uncovered during testing, as a result of the design practices employed. For lack of supporting data, however, we can only conclude that the cost benefits of these design practices may not be evident until a software system is in Operation and Maintenance.

4.5 CONCLUSIONS

The results of our data analysis show that the use of Modern Programming Practices does significantly reduce the costs of software development. In addition, our results support the hypothesis that resource expenditures are redistributed amongst the major development activities, i.e. that costs for definition and design become a larger portion of the total development cost, and costs for construction and testing become smaller.

Interpreting our data on implementation of Modern Programming Practices, we were able to detect a verifiable cost benefit for those practices which directly influence the manner in which the Program Manager's responsibilities are discharged. For larger projects, MPP related to testing activities

have a verifiable cost benefit. Configuration Management and Change Control practices appear to have a significant influence on the ease of communication and degree of involvement of all participants in a software development, which aids cost control, even though the data supporting this is inconclusive. We encountered universal and very positive evaluations of modern design practices in our project interviews, but were not able to discern a cost benefit attributable to these practices.

The remainder of the Modern Programming Practices described in Section 3 were not found by this study to have discernable effects on development costs. However, these practices may significantly influence product quality, particularly as it is reflected in responsiveness to the customer's requirements. The discipline and control inherent in MPP may also reduce risk to schedule, especially in the critical testing stages of software development. In addition, an examination of the costs of operation, maintenance, and enhancement of software developed using MPP may reveal significant cost benefits for practices whose effect on development costs were not discernible.

5.0 COMPARISON OF BCS MPP WITH IBM SPS

This section presents a comparison of the Modern Programming Practices described in Section 3.0 of this report with that set of practices described in the fifteen-volume Structured Programming Series (SPS) developed for the Air Force by IBM Corporation (A.F. contract F3060274-C-0186).

The SPS description of the software development process (see Volume VII, Figure 2-01; Appendix C, page C-8) corresponds closely to the development lifecycle depicted in Figure 3-2 of this report. Specifically, the Definition phase we describe is similar to the SPS phase called System Definition; our Design phase incorporates the SPS Design phase and the detailed design activity included in the SPS implementation phase. The MPP concept of a Construction phase includes the code and testing activities of the SPS Implementation phase, and also includes functional testing as a distinct activity. The MPP Demonstration phase includes preparation for acceptance and delivery, and formal demonstration; the Final Evaluation phase of the SPS process includes functional testing activities and is not deliberately structured for product delivery.

In comparing the practices employed in the software development process, we have developed a series of tables detailing the characteristics of similar practices. These figures, with supporting discussion, are presented in the following subsections.

5.1 PROJECT ORGANIZATION AND MANAGEMENT PROCEDURES

We found considerable similarity between the role of the Program Manager (Section 3.1.1) and the Chief Programmer (SPS Volume X). While the SPS documents are not explicit about reviews at the end of development phases (Section 3.1.2), we were able to find some similarity between the practice of Design Reviews (SPS Volume XV) and MPP Formal Reviews.

5.1.1 Program Manager

The functions and responsibilities of the Program Manager for a software development are virtually identical to those of a Chief Programmer, with one key difference. The Program Manager does not, ordinarily, perform in a technical role on the project, i.e. he does not design, code, or test software elements himself. Rather, he plans and assigns these tasks to programmers who report to him. For this reason, the qualifications of a Program Manager are defined in terms of his ability to plan, schedule, review, and control development activities in a managerial capacity, rather than in terms of his technical knowledge and programming expertise. Further, the roles of backup programmer, program librarian, and administrative assistant, as supporting members of a software development team, are not explicitly defined in MPP. (In practice, the program librarian role is frequently instituted, particularly as the designated focal point for problem reporting and resolution as discussed in Section 3.7.2.) Figure 5-1 summarizes the comparison of the Program Manager role with that of the Chief Programmer.

	Modern Programming Practices	Structured Programming Series	
	Program Manager (Reference Section 3.1.1)	Chief Programmer	Reference
Functions			
Planning	Yes	Yes	X, p.2-4
Assignment	Yes	Yes	X, p.A-3
Reviewing	Yes	Yes	X, p.A-3
Estimating	Yes	Yes	X, p.A-2,A-4
Directing	Yes	Yes	X, p.A-4
Controlling	Yes	Yes	X, p.A-3,A-4
Responsibilities			
Administrative	Yes	Yes	X, p.A-2
Product	Yes	Yes	X, p.A-2
Quality	Yes	Yes	X, p.A-5
Schedule	Yes	Yes	X, p.A-3
Cost	Yes	Yes	X, p.A-3
Profit	Yes	No	
Technical Role	No	Yes	X, p.A-3,2-3
Design	No	Yes	X, p.A-3,2-3
Code	No	Yes	X, p.A-3,2-3
Test	No	Yes	X, p.2-3
Determining Project Audit Needs	Yes	Yes	X, p.A-5
Interfaces			
Contracts	Yes	No	
Finance	Yes	No	
Customer	Yes	Yes	X, p.A-3
Line Manager	Yes	Yes	X, p.A-2,2-6
Staff	Yes	No	
Program Personnel	Yes	Yes	X, p.A-2
Backup Programmer	No	Yes	X, p.A-6
Program Librarian	No	Yes	X, p.A-9
Administrative Assistant	No	Yes	X, p.A-15
Qualifications			
Managerial Knowledge	Yes	Yes	X, p.A-4
Technical Knowledge	No	Yes	X, p.A-4
Application Knowledge	No	Yes	X, p.A-4
Decision Making - Technical	No	Yes	X, p.A-5
Decision Making - Managerial	Yes	Yes	X, p.A-5
Creative/Analytic Ability	No	Yes	X, p.A-5
Specified Experience Level	No	Yes	X, p.A-5

FIGURE 5-1
PROJECT ORGANIZATION AND MANAGEMENT PROCEDURES

5.1.2 Formal Reviews

The SPS does not explicitly define formal milestone reviews to be held at the end of each phase of software development. However, the Design Review Practice (SPS Volume XV) does entail customer involvement in a review of the design approach for satisfaction of requirements and consistency. The objectives and reviewable items for Critical Design Review and Physical Configuration Audit/Functional Configuration Audit found in Section 3.1.2 are not addressed in the SPS documents. Figure 5-2 summarizes the comparison of the MPP Formal Reviews with the SPS Design Review Practice.

	Modern Programming Practices	Structured Programming Series	
	Formal Reviews (Reference Section 3.1.2)	Design Reviews	Reference
Preliminary Design Review	Yes	Yes	XV, p.3-6
Identified Requirements	Yes	No	
Demonstration Plan	Yes	No	
Cost/Benefit Analysis	Yes	No	
User Guide(Product Specification)	Yes	No	
Major Functional Components	Yes	No	
Discuss Approach and Consistency	Yes	Yes	XV, p.3-6
Critical Design Review	Yes	No	
Detailed Construction Plan	Yes	No	
Test Plan	Yes	No	
Identified End Items	Yes	No	
Training			
Installation			
Operation			
Conversion			
.			
.			
.			
PCA/FCA			
All Identified End Items Produced	Yes	No	
All Required Functions Demonstrated	Yes	No	
Configuration Index	Yes	No	

FIGURE 5-2

PROJECT ORGANIZATION AND MANAGEMENT PROCEDURES

5.2 DOCUMENTATION STANDARDS

We found similarities between the types of documents defined in MPP (Section 3.2.1) and the SPS (Volume VII); there were some differences apparent in the scheduling of document preparation during the development process. The MPP concept of a Unit Development Folder (Section 3.2.2) is similar to the Project Notebook (SPS Volume I).

5.2.1 Document Pertinence

The classes and content of the documents identified in Volume VII of the SPS are similar to those described in Section 3.2.1. The MPP cost/benefit analysis document, produced in support of the requirements document and product specification during the Definition phase, is not explicitly mentioned in the SPS volumes. MPP requirements and design documents include specifications for data bases, file formats, etc., where the SPS identifies data requirements and specifications as a separate document. While the SPS specifies form, content, and intended audience for identified documents, it does not explicitly prescribe that the intended audience review the documents, as they are produced, for adequacy.

While MPP schedules User Guide availability at the end of Definition, and Test Plan availability at the end of Design, the SPS calls for these documents to be available at the end of the Implementation (Construction) phase. Figure 5-3 summarizes this comparison.

	Modern Programming Practices	Structured Programming Series	
	Document Pertinence (Reference Figure 3-2, Section 3.2)	Documents Produced	Reference
Identified Documents			
Requirements Document	Yes	Yes	VII,p.3-4
User Guide(Product Specification)	Yes	Yes	VII,p.3-4
Design Document	Yes	Yes	VII,p.3-4
Cost/Benefit Analysis	Yes	No	
Test Documentation	Yes	Yes	VII,p.3-4
Installation Document	Yes	No	
Operation Document	Yes	Yes	VII,p.3-4
Training Document	Yes	No	
Data Requirements and Specifications	No	Yes	VII,p.3-4
Program Maintenance Manual	Yes	Yes	VII,p.3-4
Acceptance Test Report	Yes	Yes	VII,p.3-6
Defined Audience, Form, Content	Yes	Yes	VII,p.3-7
Audience Review	Yes	No	
Scheduling of Documents			
During Definition			
Requirements Document	Yes	No	
User Guide (Product Specification)	Yes	No	
Cost/Benefit Analysis	Yes	No	
Functional Description	Yes	Yes	Vol.VII,p.3-8
Data Requirements	Yes	Yes	Vol.VII,p.3-8
During Design			
Design Document (System, subsystem & data)	Yes	Yes	Vol.VII,p.3-8
Test Plan	Yes	No	
During Construction			
Installation Document	Yes	No	
Operation Document	Yes	Yes	Vol.VII,p.3-9
Training Documents	Yes	No	
Program Maintenance	Yes	Yes	Vol.VII,p.3-9
User Guide	No	Yes	Vol.VII,p.3-10
Test Plan	No	Yes	Vol.VII,p.3-9
After Delivery			
Acceptance Test Report	Yes	Yes	Vol.VII,p.3-10

FIGURE 5-3
DOCUMENTATION STANDARDS

5.2.2 Unit Development Folder (UDF)

The key difference between the UDF as described in Section 3.2.2 and the Project Notebook defined in Volume I of the SPS is that the UDF is implemented as a project visibility tool. The cover sheet of the UDF details the specific products of tasks, the schedule and resource estimates for their completion, and provides entries for completion dates and review signatures of each product. This mechanism permits the Program Manager to make more accurate assessments of status, progress, and performance. The comparison between the UDF and the Project Notebook is summarized in Figure 5-4.

	Modern Programming Practices	Structured Programming Series	
	Unit Development Folder (Reference Section 3.2.2)	Project Notebook	Reference
Requirements	Yes	Yes	I, p.2-5
Design	Yes	Yes	I, p.2-5
Design Verification	Yes	No	
Test Plan	Yes	Yes	I, p.2-5
Test Data	Yes	Yes	I, p.2-5
Test Procedures	Yes	Yes	I, p.2-5
Test Results	Yes	Yes	I, p.2-5
Interface Descriptions	Yes	Yes	I, p.2-5
Code Listings	Yes	Yes	I, p.2-5
Code Verification	Yes	Yes	I, p.2-5
Cover Sheet	Yes	No	
Resource Estimates	Yes	No	
Schedule	Yes	No	

FIGURE 5-4
DOCUMENTATION STANDARDS

5.3 DESIGN METHODOLOGY

In comparing the design methodologies described in the SPS with those of Section 3.3 of this report, we found considerable similarities in the practice of top down design, and in the techniques employed to verify designs. The primary distinction between MPP design practices and those of the SPS was the deliberate separation of the design activity from coding and checkout activities.

5.3.1 Design Completion

The phasing of design activities within the software development process, as described in Section 3.1 of this report and in the SPS (Volume VII), is noticeably different. The modern practice of Design Completion (see Section 3.3.1) implies that the design of software elements is distinct from and completed prior to the construction activities (coding and checkout). Further, the design is subjected for formal review prior to the beginning of construction, and the results of the design activity form the basis for a detailed construction plan. In the SPS description, design proceeds in parallel with construction activities as part of the Implementation phase, and no formal design review is called for. Figure 5-5 summarizes this comparison.

	Modern Programming Practices	Structured Programming Series	
	Design Completion (Reference Section 3.3.1)	Design Completion	Reference
Design Documented	Yes	Yes	VII,p.3-7
Complete Before Coding	Yes	No	
Detail Sufficient For Construction Planning	Yes	No	
Approved At CDR For Coding	Yes	No	

FIGURE 5-5
DESIGN METHODOLOGY

5.3.2 Design Verification

Design verification via structured walkthrough is a key element of the methodologies described in this report (Section 3.3.2) and in the SPS (Volume VIII). However, the MPP structured walkthrough is implemented as a one-on-one review, between the designer and a peer, a lead programmer or a customer technical representative, whereas the SPS structured walkthrough may be conducted as a many-on-one review, involving the designer and a team of reviewers. The MPP structured walkthrough is not used as a device to teach new programmers or to afford management visibility of the details of the design; rather, it concentrates on analysis of the design for consistency and satisfaction of requirements. The comparison between MPP Design Verification and the SPS practices of Program Review and Design Validation is summarized in Figure 5-6.

	Modern Programming Practices	Structured Programming Series	
	Design Verification (Reference Section 3.3.2)	Program Review and Design Validation	Reference
Structured Walkthrough	Yes	Yes	VIII,p.6-15
Individual Review	Yes	Yes	VIII,p.6-16
Group Review	No	Yes	VIII,p.6-16
Verify Requirements Satisfied	Yes	Yes	XV,p.3-20
Formal Design Representation	Yes	No	VIII,p.7-2
Signoff	Yes	Yes	XV,p.3-22
Design Consistency	Yes	Yes	XV,p.3-20
Reachability and Connectivity Analysis	Yes	No	
Direction of Design Errors	Yes	Yes	VIII,p.6-15
Teaching New Programmers	No	Yes	VIII,p.6-15
Communication of Interfaces	Yes	Yes	VIII,p.6-15
Management Visibility	No	Yes	VIII,p.6-15

FIGURE 5-6
DESIGN METHODOLOGY

5.3.3 Top Down Design

The practice of top down design as described in Section 3.3.3 and in the SPS (Volumes I and VIII) are very similar. One key aspect of MPP top down design is the emphasis on incorporation of available, off-the-shelf software elements as the design is decomposed. The design representation techniques employed in MPP are symbolic and formalized, whereas the SPS advocates an English-like Program Design Language. The comparison between MPP top down design and that described in the SPS is summarized in Figure 5-7.

	Modern Programming Practices	Structured Programming Series	
	Top Down Design (Reference Section 3.3.3)	Top Down Design	Reference
Decomposition of Requirements	Yes	Yes	I,p.2-5
Testing For Correctness, Workability Level-By-Level	Yes	Yes	I,p.2-4
Requirements Satisfied At each Level	Yes	Yes	I,p.2-5
Top Down Checkout	Yes	Yes	I,p.2-4
Top Down Design Verification	Yes	Yes	VIII,p.6-15
Top Down Definition To Use Available Software Routines	Yes	No	
Design To Level of Primitives	Yes	Yes	VIII,p.2-4
Design Representation			
Flowcharts	Yes	No	VII,p.4-2
HIPO	Yes	Yes	VIII,p.6-1
Transition Diagrams	Yes	No	
Design Trees	Yes	Yes	VIII,p.6-3
English-Like Program Design Language	No	Yes	VIII,p.7-2

FIGURE 5-7
DESIGN METHODOLOGY

5.4 PROGRAMMING STANDARDS

The programming standards defined in the SPS (Volume I) are virtually identical to those incorporated in Modern Programming Practice. While MPP coding conventions do not explicitly define such standards as single entry/exit, use of the 'include' capability for incorporating common blocks of code, specifications for statement numbering, continuation cards, and definitions of statement types, these standards have been part of the traditional programming practices from which MPP evolved. These standards are almost universally followed at BCS, if only because the 'standard' compilers available to software projects directly support such conventions. Figures 5-8 and 5-9 provide summaries of the comparisons of MPP and SPS structured programming and coding conventions.

The practice of code verification via peer review is also nearly identical in MPP and in the SPS; the key difference is that MPP code verification emphasizes a tangible result from the peer review, usually in the form of the reviewer's signature on the UDF cover sheet. Figure 5-10 summarizes the comparison of MPP and SPS code verification.

	Modern Programming Practices	Structured Programming Series	
	Structured Forms (Reference Section 3.4.1)	Structured Programming	Reference
Structured Logic Forms	Yes	Yes	I,p.2-2
Code Forms Correspond To Design Representation	Yes	Yes	I,p.2-4
Blocked Coding Sequences	Yes	Yes	I,p.2-3
Identification of Code	Yes	Yes	I,p.3-2
Prohibition On Violating Forms (e.g. GO-TO)	Yes	Yes	I,p.3-2
Idiomatic Substitutions For Structured Forms	Yes	Yes	I,p.4-31
All Code Segments Have Single Entry, Single Exit	No*	Yes	I,p.3-1
The Beginning and End of Any Segment Completely Contained	No*	Yes	I,p.3-2
In Free Format Languages, Only 1 Statement Per Line of Code	No*	Yes	I,p.3-2
Implementation Of The 'Include' Capability	No*	Yes	I,p.6-1

* Considered part of traditional programming standards

FIGURE 5-8
PROGRAMMING STANDARDS

	Modern Programming Practices	Structured Programming Series	
	Coding Conventions (Reference Section 3.4.2)	Coding Conventions	Reference
Naming Conventions for Code/Document Cross-Reference	Yes	No	
Higher Order Language	Yes	Yes	I,p.4-1
Code Commentary	Yes	Yes	I,p.4-42
Interface Definitions	Yes	Yes	I,p.6-1
Statement Numbering	No*	Yes	I,p.4-42
Continuation Cards	No*	Yes	I,p.4-43
Statement Specifications	No*	Yes	I,p.4-43

* Considered part of traditional programming standards

FIGURE 5-9
PROGRAMMING STANDARDS

	Modern Programming Practices	Structured Programming Series	
	Code Verification (Reference Section 3.4.3)	Peer Review	Reference
Peer Inspection	Yes	Yes	XV,p.5-3
Prior To Checkout	Yes	Yes	XV,p.5-5
After Checkout	Yes	Yes	XV,p.5-4
Review Checkout Results	Yes	Yes	XV,p.5-4
Signoff	Yes	No	

FIGURE 5-10
PROGRAMMING STANDARDS

5.5 SUPPORT LIBRARIES AND FACILITIES

The BCS-developed design aid, structured precompiler, and programming support library aid offer some of the features defined in the SPS specifications for such tools. It should be noted that the SPS describes aids that do not yet exist, whereas the MPP aids are implemented and in use. Some of the requirements for the MPP aids which exist are different from those documented in the SPS. In addition, while such tools as compilers, subroutine libraries, operating system services, and utility programs were discussed in Section 3.5, we have not included them in our comparisons.

5.5.1 Design Aids

The BCS automated design aid (see Section 3.5.1) provides support to design documentation similar to that specified for the SPS Programming Support Library (PSL). In addition, the design provides basic design verification functions. MPP manual aids for the design activity differ from those described in the SPS primarily because of the MPP emphasis on a more symbolic design language, versus the SPS-defined English-like Program Design Language. Figure 5-11 summarizes this comparison.

	Modern Programming Practices	Structured Programming Series	
	Design Aids (Reference Section 3.5.1)	Programming Support Library and Design Aids	Reference
Automated Aid			
Design Representation	Yes	Yes	V,p.(2-)5
Functions	Yes	No	
Interfaces	Yes	No	
Input/Output	Yes	No	
Transition Conditions	Yes	No	
Design Verification	Yes	No	
Transition Conditions	Yes	No	
Input/Output	Yes	No	
Exit Checking	Yes	No	
Manual Aids			
Document Prototypes	Yes	No	
Procedural Forms	Yes	No	
Design Analysis Algorithms	Yes	Yes	XV,p.3-19
HIPO	Yes	Yes	VIII,p.7-1
English-Like Program Design Language	No	Yes	VIII,p.7-1

FIGURE 5-11
SUPPORT LIBRARIES AND FACILITIES

5.5.2 Structured Precompiler

The BCS structured precompiler for Fortran (see Section 3.5.2) implements the features specified for such an aid in the SPS (Volume II). The MPP manual aids to support structured coding correspond to those discussed in the SPS. Figure 5-12 summarizes the comparison of MPP aids for structured coding with those defined in the SPS.

	Modern Programming Practices	Structured Programming Series	
	Structured Precompiler (Reference Section 3.5.2)	Structured Precompiler	Reference
Automated Aid			
Accepts Structured Forms	Yes	Yes	II,p.2-1
Translates to HOL	Yes	Yes	II,p.2-2
Closure and Nesting Checking	Yes	Yes	V,p.(3-)12
Statement Identification	Yes	No	
Block Identification	Yes	No	
Enriched Structured Forms	Yes	Yes	I,p.B-1
Input Specification			
IFTHENELSE	Yes	Yes	II,p.3-11
DOWHILE/DOUNTIL	Yes	Yes	II,p.3-13
CASE	Yes	Yes	II,p.3-15
Output Specification			
IFTHENELSE	Yes	Yes	II,p.4-6
DOWHILE/DOUNTIL	Yes	Yes	II,p.4-6
CASE	Yes	Yes	II,p.4-7
Manual Aid			
Code Verification by Peer Inspection	Yes	Yes	XV,p.5-3
Prescribed Logic Forms	Yes	Yes	I,p.3-2
Proscribed Logic Forms	Yes	Yes	I,p.3-2,p.3-17

FIGURE 5-12
SUPPORT LIBRARIES AND FACILITIES

5.5.3 Programming Support Library Aid

The BCS programming support library aid provides the support for recording of coded elements and for file maintenance specified in Volume V of the SPS. It does not supply the bulk of the management information specified in the SPS definition of the programming support library aid (and re-iterated in Volume IX, Management Data Collection and Reporting). In particular, for the visibility the Program Manager requires for assessment of status, progress, and performance, we believe that the plan data and actual data specified in Volumes V and IX is particularly critical. Further, the program maintenance statistics (specified in SPS Volume V) and the MPP manual ledgers (configuration index) of software elements appear to be of significant benefit in supporting configuration management and change control. The comparison of features of BCS' programming support library aid with the specifications in the SPS is summarized in Figure 5-13.

Automated Aid	Modern Programming Practices: Programming Support Library Aid (Reference Section 3.5.3)	Structured Programming Series: Programming Support Library Aid and Management Data Collection and Reporting	Reference
Recording of			
Designs	Yes	Yes	V,p.(3-)8
Source Code	Yes	Yes	V,p.(3-)8
Object Code	Yes	Yes	V,p.(3-)8
Data Cases	Yes	Yes	V,p.(3-)8
Job Control Language	Yes	Yes	V,p.(3-)8
Macro Code	Yes	Yes	V,p.(3-)8
Load Modules	Yes	Yes	V,p.(3-)8
Textual Data	Yes	Yes	V,p.(3-)8
File Maintenance			
Copy	Yes	Yes	V,p.(3-)9
Delete	Yes	Yes	V,p.(3-)9
Add	Yes	Yes	V,p.(3-)9
Update	Yes	Yes	V,p.(3-)9
Merge	Yes	Yes	V,p.(3-)10
Compression	Yes	Yes	V,p.(3-)10
Management Information			
Lists Identifiers and Entry Dates	Yes	Yes	V,p.(3-)18
Collection and Reporting Levels	No	Yes	V,p.(3-)18
Plan Data	No	Yes	V,p.(3-)8
Actual Data	No	Yes	V,p.(3-)16
Archive Data	No	Yes	V,p.(3-)17
Program Module Statistics	No	Yes	V,p.(3-)18
Computer Utilization Rpts.	No	Yes	V,p.(3-)18
Program Maint. Statistics	No	Yes	V,p.(3-)18
Program Structure Reports	No	Yes	V,p.(3-)18
Historical Reports	No	Yes	V,p.(3-)18
Combinational, Cyclical & Exceptional Reports	No	Yes	V,p.(3-)19
Manual Aids			
Ledgers For Elements	Yes	No	
Name	Yes	No	
Content	Yes	No	
Location	Yes	No	
Version	Yes	No	
Relationships	Yes	No	

FIGURE 5-13
SUPPORT LIBRARIES AND FACILITIES

5.6 TESTING METHODOLOGY

The Modern Programming Practices applied to (software and system) testing incorporate many of the characteristics of the testing methodology described in the SPS. MPP techniques emphasize a more formal acceptance and demonstration activity, encourage the use of "live" data (supplied by the customer), and stress detailed planning of all testing activities.

5.6.1 Acceptance Testing

The modern practices applied to acceptance testing (see Section 3.6.1) include a "dress rehearsal" of the demonstration, resulting in control listings which are then used during the formal demonstration; these techniques are not discussed in the SPS. The characteristics of acceptance testing described in (Volume XV of) the SPS are incorporated in the MPP methodology. The MPP emphasis on system operability, installability, and the use of customer-supplied usability exercises is not specified in the SPS description of acceptance testing. The comparison of MPP and SPS acceptance testing is summarized in Figure 5-14.

	Modern Programming Practices	Structured Programming Series	
	Acceptance Testing (Reference Section 3.6.1)	Acceptance Testing	Reference
Confirm Operation Readiness	Yes	No	
Formal Demonstration	Yes	Yes	XV,p.3-3
Test Requirements	Yes	Yes	XV,p.3-3
Test Plan	Yes	Yes	XV,p.3-3
Installability	Yes	No	
Operability	Yes	Yes	XV,p.3-3
Usability (User Data)	Yes	No	
Dress Rehearsal	Yes	No	
Control Listings	Yes	No	
Plans Reviewed At CDR	Yes	Yes	XV,p.3-3

FIGURE 5-14
TESTING METHODOLOGY

5.6.2 Functional Testing

Functional testing practices described in Section 3.6.2 of this report and in the SPS (Volume XV) are quite similar. MPP functional testing encourages the use of customer-supplied data, which is not mentioned in the SPS. In addition, since functional testing is usually performed by individuals not authorized to make code changes when problems are discovered, a problem reporting and resolution procedure is specified in MPP as a necessary supporting activity. The comparison of MPP and SPS approaches to functional testing is summarized in Figure 5-15.

	Modern Programming Practices	Structured Programming Series	
	Functional Testing (Reference Section 3.6.2)	Functional Testing	Reference
Verify All Functions	Yes	Yes	XV,p.3-17
Reasonableness Testing	Yes	Yes	XV,p.3-17
Test Plans	Yes	Yes	XV,p.3-17
Test Procedures	Yes	Yes	XV,p.3-17
Test Data ("Live")	Yes	No	
Independent Team Conducts Tests	Yes	Yes	XV,p.3-17
Problem Reporting and Resolution	Yes	No	
Customer Inspects Results For Reasonableness	Yes	Yes	XV,p.3-18

FIGURE 5-15
TESTING METHODOLOGY

5.6.3 Test Formalism

The modern practice of test formalism (see Section 3.6.3) incorporates some of the practices cited in the SPS document on Validation and Verification (Volume XV). In addition, test formalism stresses the review of test plans, test designs, and the schedule of test activities with the customer. Figure 5-16 summarizes the comparison of MPP and SPS formalized testing practices. It should be noted that many of the specific testing techniques described in the Validation and Verification volume of the SPS have been used on BCS software development projects. In our study, no attempt was made to identify which techniques were applied on the projects we investigated; rather, we chose to concentrate on the formality with which these techniques were implemented. Our comparison reflects this emphasis.

	Modern Programming Practices	Structured Programming Series	
	Test Formalism (Reference Section 3.6.3)	Validation and Verification and Chief Programmer	Reference
Test Plans	Yes	Yes	VII,p.3-9
Necessary and Sufficient	Yes	Yes	XV,p.3-18
Complete	Yes	No	
Correct	Yes	No	
Operable	Yes	Yes	XV,p.3-3
Test Specifications	Yes	Yes	XV,p.3-18
Preparation of Procedures	Yes	Yes	XV,p.3-6
Preparation of Data	Yes	Yes	XV,p.3-5
Determination of Results	Yes	No	
Resource Estimates	Yes	Yes	X,p.A-3
Task Assignments	Yes	Yes	X,p.A-3
Use of "Live" Data	Yes	No	XV,p.3-5
Sequenced Top Down	Yes	Yes	XV,p.5-1
Control of Test Materials	Yes	Yes	XV,p.B-5
Test Plans Reviewed At PDR	Yes	No	
Test Designs and Schedule Reviewed At CDR	Yes	No	

FIGURE 5-16
TESTING METHODOLOGY

5.7 CONFIGURATION MANAGEMENT AND CHANGE CONTROL

The SPS does not address the issue of a Change Control Board (see Section 3.7.3) as a constituted entity in a software development project. The SPS definition of the Chief Programmer's responsibilities (Volume X) prescribes that individual as the focal point for problem reporting and resolution (similar to discussion in Section 3.7.2), but does not detail the procedures used to carry out that function.

The Programming Support Library Aid (Volume VI) specified by the SPS provides support for baselining activities, but does not support the unique identification of constituents to specified capabilities or implementations, as described in Section 3.7.1. A summary of the comparison of MPP configuration management and change control with practices described in the SPS is presented in Figures 5-17, 5-18, and 5-19.

	Modern Programming Practices	Structured Programming Series	
	Baselining (Reference Section 3.7.1)	Program Support Library and Indexes (Directories)	Reference
Baseline Documentation	Yes	Yes	VI,p.2-3
Configuration Index	Yes	Yes	VI,p.2-4
Capabilities List	Yes	No	
Constituents List	Yes	No	
Implementation List	Yes	No	
Baseline Qualifiers	Yes	Yes	VI,p.2-4
System	Yes	Yes	VI,p.2-4
Version	Yes	Yes	VI,p.2-4
Release	Yes	No	
Facility	Yes	No	
Revision	Yes	Yes	VI,p.2-3

FIGURE 5-17
CONFIGURATION MANAGEMENT AND CHANGE CONTROL

	Modern Programming Practices	Structured Programming Series	
	Problem Reporting and Resolution (Reference Section 3.7.2)	Chief Programmer and User/Customer Relationship	Reference
Focal Point	Yes	Yes	X,p.A-3
Report Problems	Yes	Yes	X,p.A-3
Resolution Status	Yes	Yes	X,p.A-3
Resolution Feedback	Yes	Yes	X,p.A-3
Problem Closeout	Yes	No	
Problem Reporting	Yes	No	
Receive Problems	Yes	No	
Gather Evidence	Yes	No	
Identification Code	Yes	No	
Receiving Resolution Status	Yes	No	
Resolution Feedback	Yes	No	
Problem Accepted/Rejected	Yes	No	
Problem Corrected	Yes	No	
Problem Closeout	Yes	No	
Change Proposal Accepted	Yes	No	
Change Proposal Rejected	Yes	No	
Change Incorporated In Revision	Yes	No	

FIGURE 5-18
CONFIGURATION MANAGEMENT AND CHANGE CONTROL

	Modern Programming Practices	Structured Programming Series	
	Change Control Board (Reference Section 3.7.3)	(None)	Reference
Board Members	Yes	No	
Know Customer's Mission	Yes	No	
Know System Use	Yes	No	
Represent			
User	Yes	No	
Developer	Yes	No	
Operator	Yes	No	
Sponsor	Yes	No	
Make Commitments	Yes	No	
Chaired By Developer Or Customer	Yes	No	
Act On Change Proposals	Yes	No	
Problem Review	Yes	No	
Problem Prioritization	Yes	No	
Change Funding and Schedule	Yes	No	
Action Items	Yes	No	
Approve Releases	Yes	No	
Specify Changes Incorporated	Yes	No	
Release Date	Yes	No	
Action Ideals	Yes	No	

FIGURE 5-19
CONFIGURATION MANAGEMENT AND CHANGE CONTROL

5.8 COMPARISON SUMMARY

In comparing the practices documented in the SPS with those modern practices in use at BCS, we found considerable similarity in most of the practices. The key differences we found are summarized below:

- Unlike the role defined for the Chief Programmer, a software Program Manager typically does not perform key technical tasks such as designing, coding, and testing. Rather, he is responsible for planning these tasks, assigning them to project personnel, and task monitoring and control.

- The disciplines of software configuration management and change control are an integral part of the modern practices in use at BCS, but are not included in the practices documented in the SPS.

- The formality of MPP testing disciplines (particularly as they are applied to acceptance testing) is significantly greater than that described in the SPS.

6.0 RECOMMENDATIONS

The conclusions of this study lead us to recommend that Air Force software procurements should be structured to encourage the use of certain beneficial Modern Programming Practices. Specifically, we recommend that particular emphasis be given to management planning and visibility, to disciplined testing, to configuration management and change control, and to disciplined top down design. However, care must be taken to preserve those aspects of the procurement environment which foster competition. Our recommendations concerning software procurement are detailed in Section 6.1.

Our study identified several other areas which we feel should be the subject of further research. Some of these recommendations were introduced in Section 1.5; these and other specific recommendations are discussed in Section 6.2.

6.1 SOFTWARE PROCUREMENT GUIDELINES

While certain software development practices may have a beneficial effect on costs, schedule risk, and product quality, it must remain the contractor's responsibility to choose and employ the practices he believes will be most cost effective. For a specific software procurement, several contractors may be qualified to do the work, by reason of their past experience and knowledge of the customer's application. These contractors may also have the necessary equipment, facilities, tools, and data to perform the work. What distinguishes among them is the methods, techniques, and disciplines each contractor imposes on the software development process to maximize cost benefits. For this reason, we believe that it is inappropriate for the customer to specify what methods, techniques, and disciplines must be employed. On the other hand, we believe that it is both appropriate and necessary that the customer specify desired qualities and characteristics of the end items to be produced. In doing so, the customer can encourage the use of beneficial practices.

The emphasis discernable in the Modern Programming Practices investigated in this study is on focus; software development activities are focused and identifiable. As a result, they are more manageable (i.e. controllable). Key to achieving that focus is the concentration of software development responsibility. We feel it is critical that, for both "embedded" and "deliverable" software capabilities, the contractor be required to focus the responsibility for software development with functions and duties similar to those we described for a Program Manager (see Section 3.1).

An additional factor which ensures the appropriate focus on software development activities is the identification (and detailed planning for the development) of those specific products which result from the activities. Focusing on tangible results permits improved management and customer visibility of the development process, and allows a more objective evaluation of status, progress, and performance.

We believe that existing military standards and specifications are sufficiently comprehensive to encourage the necessary focus on software development activities and products. For the most part, the provisions of these standards and specifications appear to encourage the use of those practices determined in this study to have significant cost benefits in software development.

6.1.1 Applicability of MIL-S-52779

Specifically, we believe that the cost benefits cited in our analysis can be achieved if the customer and the software Program Manager adhere to the provisions of Military Specification 52779,[1] included in Appendix A of this report for reference. This specification states that the contractor must supply, in his Quality Assurance Plan, descriptions of the practices he will use in developing a computing capability. The provisions of paragraphs 3.2.1 through 3.2.4, 3.2.6, and 3.2.8 directly relate to the beneficial practices we studied.

Paragraph 3.2.1, on Work Tasking and Authorization Procedures, addresses the issue of detailed task planning and formal, written task assignments (see Section 3.1). Paragraph 3.2.3, on Configuration Management, and paragraph 3.2.4, on Corrective Action, refer to those procedures necessary to support software configuration management and change control (see Section 3.7). The formalized approach to software test planning and conduct (see Section 3.6) is appropriately addressed in paragraph 3.2.3. Computer Program Design, paragraph 3.2.6, establishes the basis for the practice of design verification (see Section 3.3). Finally, paragraph 3.2.8, on Reviews and Audits, refers to the formal reviews (PDR, CDR, and PCA/FCA) discussed in Section 3.1.2 of this report.

In the NOTES section of this military specification, we feel that the referenced Military Standards 483[2] and 490[3] are appropriate. However, the referenced Military Standard 1521[4] is only partially appropriate within the context of Modern Programming Practices.

---

[1] MIL-S-52779 (AD), dated 5 April 1974, "Software Quality Assurance Program Requirements".

[2] MIL-STD-483, dated 1 June 1971, "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs".

[3] MIL-STD-490, dated 18 May 1972, "Specification Practices".

[4] MIL-STD-1521A, dated 1 June 1976, "Technical Reviews and Audits for Systems, Equipments, and Computer Programs".

6.1.2 Applicability of MIL-STD-1521

We believe that the definitions of Functional Configuration Audit (FCA), Physical Configuration Audit (PCA), and Formal Qualification Review (FQR) are, to a great extent, consistent with Modern Programming Practices. The definitions of the objectives and conduct of Preliminary Design Review (PDR) and Critical Design Review (CDR) appear inappropriate in the context of MPP.

Specifically, we believe that appropriate system requirements and sub-elements defined as review items at System Design Review (see Appendix B of MIL-STD-1521A) should be re-examined at a software Preliminary Design Review (PDR). In addition, Sections 30.5 through 30.17 of Appendix C (of MIL-STD-1521A) address many aspects which we believe are pertinent to a software PDR, but are expressed only in hardware terms. We believe these paragraphs need to be restated in appropriate software terminology to make their objectives and applicability for software more obvious.

Appendix D (of MIL-STD-1521A) describing Software Critical Design Review is largely inappropriate for the conduct of a formal review within the context of MPP. The purpose of a CDR should be to confirm that the identified elemental products of the (planned) construction process are both necessary and sufficient to satisfy the requirements formally established at PDR. The emphasis should be on item identification (via a configuration index) and plans for construction, rather than on the integrity of design detail. The contractor should provide (at CDR) evidence that design details have already been reviewed (perhaps by customer technical representatives), so that the formal review need not itself address those details.

The discussion of Functional Configuration Audit (Appendix E of MIL-STD-1521A) appears appropriate as written. We recommend that an explicit statement be added which stipulates that the contractor performs the activities described, and provides evidence at the formal FCA that he has done so. To facilitate this, we recommend that the distinction be clearly drawn between test procedures and test minutes (the essential evidence of FCA). Test minutes may be simply a copy of a test procedure in which results necessary to determine test outcome have been recorded. The evidence submitted at FCA should show that test minutes exist for each test, have been appropriately reviewed, and are in the possession of a designated individual.

The discussion of Physical Configuration Audit (Appendix F of MIL-STD-1521A) is appropriate for software PCA. The key objective of a software PCA should be the identification of all "software parts", at least to the level of subroutines and document sections. The evidence of changes introduced (or proposed but not acted upon) since PDR should be a direct outgrowth of the contractor's on-going configuration management and change control activities (see MIL-S-52779, paragraphs 3.2.2 and 3.2.4).

We believe it is inappropriate for a software Formal Qualification Review to be combined with FCA, as suggested in the discussion of Formal Qualification Review (Appendix G, paragraph 70.2 of MIL-STD-1521A). The formal demonstration of a software capability (including installability, operability,

and usability demonstrations) should be a pre-condition of certification at FQR. That demonstration should be conducted after the successful completion of PCA/FCA, and should be performed at the receiving site.

6.1.3 Summary

The use of beneficial Modern Programming Practices can be encouraged by the customer via deliberate application of existing military specifications, especially MIL-S-52779. Certain other military standards and specifications, notably MIL-STD-1521, are only partially applicable within the context of MPP, as they currently exist. The Air Force should examine existing military specifications and standards to assess their applicability to MPP software developments and modify them as appropriate.

6.2 RECOMMENDATIONS FOR FURTHER STUDY

As a result of this study, several areas have been identified where further investigation or development seems appropriate. Specific recommendations are detailed in the following paragraphs.

6.2.1 Management Guidance on MPP Application

The significance of management's role in achieving maximum benefits from the application of Modern Programming Practices implies that specific guidance for managers of software development is needed. Command media should be developed to address how appropriate MPP can be chosen based on project objectives and activities. The management disciplines associated with planning, task assignment, and monitoring within a project employing MPP should be examined and formalized into improved specifications and guidelines for software development.

6.2.2 MPP Impact on Product Quality

While this study did not directly address the issue of quality of the delivered software product, Modern Programming Practices stress the concept of quality, particularly as it is reflected in demonstrated responsiveness to user requirements. The impacts of MPP on software product quality should be determined, perhaps by studying the activity of a system from the completion of design through operational use. Such a study should consider change history in terms of repairs and enhancements, measures of customer satisfaction, and benefits expected and realized during the production life.

6.2.3 MPP Impact on Schedule Risk

The impact of Modern Programming Practices on schedule risk can be inferred from an examination of the resource expenditure data gathered in this study. Largely as a result of deliberate pre-planning, testing and other construction activities performed late in the development cycle appear to be both more manageable and less prone to schedule slippage. Directed study should be conducted to determine conclusively whether MPP-using projects incur less schedule risk than software projects which use traditional methods.

6.2.4 MPP Impact on Lifecycle Costs

This study considers the effects of Modern Programming Practices on software development costs; there remains the larger question of how MPP affect software lifecycle costs. Specifically, the costs of operation, maintenance, and enhancement of a system originally developed using MPP may show considerable benefit. Those MPP whose impact is relatively minor during system development may have a more significant impact during the production life of a delivered system.

6.2.5 Cost Estimating For MPP

The development cost benefits attributable to Modern Programming Practices are significant enough that software cost estimating techniques currently in use throughout the industry are no longer completely valid. This study shows that projects employing MPP spend less resources, and spend them differently, than what traditional estimating guidelines forecast. The algorithms which the software industry uses to estimate development costs need to be adjusted by new parameters which account for the influences of specific MPP.

6.2.6 Customer Training in MPP

The changing role of the customer and the software developer, and the new practices which the developer is using to fulfill his obligations must be understood by those DoD offices responsible for software procurements. Specific training for procurement-office personnel in such areas as the objectives and conduct of formal reviews and the customer's participation in definition, design, and testing activities should be encouraged.

6.2.7 Support Tools For MPP

The opinions expressed by our projects regarding the tools currently available to support Modern Programming Practices (see Section 3.5) leads us to conclude that tools are needed, but that their capabilities must be significantly expanded beyond the level of support currently provided. An automated design aid should be developed which alleviates the dependance on the structured walkthrough for design verification; more sophisticated logic analysis and interface consistency checking capabilities should be provided. A Programming Support Library aid which provides extensive management visibility information could effectively support the Program Manager's assessment of status, progress, and performance. Compilers which directly support structured logic formulations would alleviate a project's dependance on manual peer code reviews.

APPENDIX A

REFERENCED MILITARY SPECIFICATIONS

MILITARY SPECIFICATION

SOFTWARE QUALITY ASSURANCE PROGRAM REQUIREMENTS

This specification is approved for use by all Departments and Agencies of the Department of Defense.

1. SCOPE

1.1 Applicability. When referenced in the item specification, contract, or order, this specification shall apply to the acquisition of software (computer programs and related documentation) where the acquisition involves either software alone or software as a portion of a system or subsystem.

1.2 Contractual Intent. This specification requires the establishment and implementation of a Software Quality Assurance (QA) Program by the contractor. The purpose of the Software QA Program is to assure that software delivered under the contract complies with the requirements of the contract. The purpose of this specification is to assure that the program is effective, economical, and planned and developed in consonance with the contractor's other administrative and technical programs. The term "Software QA Program", as used herein identifies the collective requirements of the specification. The Software QA Program may be an extension of the contractor's existing QA program. The Software QA Program shall provide for periodic assessment and, where necessary, realignment of the QA program to conform to changes in the acquisition program. The Software QA Program is subject to disapproval by the Government whenever it does not accomplish the requirements of this specification.

1.3 Relation to Other Contract Requirements. The contractor is responsible for compliance with all provisions of the contract and for furnishing specified software which meet all the requirements of the contract. If any inconsistency exists between the terms of the contract and this specification, the Order of Precedence clause shall govern.

FSC IPSC

2. APPLICABLE DOCUMENTS.

2.1 Amendments and Revisions. Whenever this specification is amended or revised subsequent to its contractually effective date, the contractor may follow or authorize his subcontractors to follow the amended or revised document provided no increase in cost, price, or fee is required. The contractor shall not be required to follow the amended or revised document except as a formally authorized modification to the contract. If the contractor elects to follow the amended or revised document, he shall notify the contracting officer in writing of this election. When the contractor elects to follow the provisions of an amendment or revision, he must follow them in full.

2.2 Ordering Government Documents. Copies of specifications, standards, and documentation required by contractors in connection with specific procurements may be obtained from the procuring agency, or as otherwise directed by the contracting officer.

3.0 REQUIREMENTS

3.1 Software QA Program. The contractor shall develop and implement a Software QA Program to assure that all software delivered meets all contractual requirements. This program shall provide for detection, reporting, analysis, and correction of software deficiencies. Contractor personnel performing quality functions shall have the responsibility and authority to evaluate software development activities, and to recommend improvements. The program, including procedures, schedules, processes, and products, shall be documented.

3.2 Software QA Program Requirements. The contractor's Software QA Program will specifically address the following areas:

3.2.1 Work Tasking and Authorization Procedures. Procedures to be used by the contractor in issuing work tasking instructions for all work relating to software development to be accomplished under the contract shall be identified and documented. The Software QA Program shall provide for monitoring the execution of these procedures to assure that they are being followed. Procedures to track progress of the work against approved schedules and resource allocations will also be identified. Areas to be addressed in these reviews shall include: the description of the work to be accomplished; assignment of responsibility for its accomplishment; the manner in which the initiation of work is authorized; the manner in which periodic status reports on the progress of work against approved schedules and resource allocations are prepared and submitted; and scheduled completion date.

3.2.2 Configuration Management (CM). The Software QA Program shall specify the quality assurance measures to be applied to CM and will provide for independent audits by contractor personnel of the contractor's CM procedures, to insure that the objectives of the CM program are being attained. The program will designate those contractor personnel responsible for conducting the audit of CM. The Software QA Program will require documentation of the audit.

3.2.3 Testing. The Software QA Program shall identify all QA measures related to software testing. Software QA measures shall include:

a. Analysis of software requirements to determine testability.

b. Review of test plans for compliance with appropriate standards and satisfaction of contractual requirements.

c. Review of test requirements and criteria for adequacy, feasibility, and satisfaction of requirements.

d. Review of test procedures for compliance with appropriate standards and satisfaction of contractual requirements.

e. Monitoring of tests and certification that test results are the actual findings of the tests.

f. Review and certification of test reports.

g. Insuring that test related documentation is maintained to allow repeatability of tests.

h. Assuring that any support software and computer hardware used to develop software for the Government are themselves acceptable to the Government. The contractor shall identify to the Government any such support software and computer hardware that he intends to use.

3.2.4 Corrective Action. The Software QA Program shall delineate those contractor procedures which shall assure the prompt detection and correction of deficiencies which otherwise could result in noncompliant software. Corrective action shall include, as a minimum:

a. Analysis of data and examination of problem and deficiency reports to determine extent and causes,

b. Analysis of trends in performance of work to prevent the development of nonconforming products.

c. Review of the adequacy of improvement on corrective measures to be taken, and monitoring of the implementation to determine effectiveness of such measures.

d. Analysis or review as otherwise provided for in the contract or by the contracting officer.

3.2.5 Library Controls. The contractor shall establish and identify, in the Software QA Program, his procedures and controls, manual or automated, for the handling of source code and object code in their various forms and versions, from the time of their initial approval or acceptance until they have been incorporated into the final deliverable media. The objective of these controls is to insure that different computer program versions are accurately identified and documented, that no unauthorized modifications are made to the source or object programs, that all approved modifications are properly integrated, and that software submitted for testing is the correct version.

3.2.6 Computer Program Design. The Software QA Program shall include the procedures by which design documentation is reviewed to evaluate the logic of the design, fulfillment of requirements, completeness, and compliance with specified standards. As a minimum, QA reviews of design documentation will be accomplished prior to release of the computer program designs to coding.

3.2.7 Software Documentation. Documentation standards to be used for all deliverable software shall be stated or referenced in the Software QA Program. The Software QA Program will specify the quality assurance measures to be applied to assure delivery of correct documentation and change information to the Government. In addition, the program will provide for the periodic independent contractor review of documentation and will designate contractor signature approval/disapproval authority.

3.2.8 Reviews and Audits. The Software QA Program shall describe or reference the contractor's procedures for preparation and execution of reviews and audits at key points in the acquisition cycle, and the quality assurance measures to be employed to insure that the reviews and audits are conducted in accordance with the prescribed procedures. The schedule for reviews and audits will be stated in the program.

3.2.9 Tools, Techniques, and Methodologies. The contractor shall identify in the Software QA Program the tools, techniques, and methodologies to be employed in the performance of the work which will support quality assurance objectives and describe how their use will augment or satisfy QA program requirements. Examples include: Operations Research - Systems Analysis techniques, modeling, simulation, software performance measurement tools and techniques, and software optimization tools.

3.3 Subcontractor Control. The contractor is responsible for assuring that all software procured from his subcontractors conform to the contract requirements. When the Government elects to perform reviews at the subcontractor's facilities, such reviews shall not be used by contractors as evidence of effective control of quality of subcontractors by the contractor. It does not relieve the contractor of his responsibility for furnishing software that meets all contract requirements.

4.0 RESPONSIBILITIES.

4.1 Contractor. Nothing specified herein relieves the contractor from the obligation to submit, to the Government for acceptance, end products that conform to all prescribed requirements.

4.2 Government Review at Contractor, Subcontractor, or Vendor Facilities. The Government reserves the right to review, at their sources, all products or services, including those not developed or performed at the contractor's facility, to determine the conformance of products or services with contract requirements.

5.0 PREPARATION FOR DELIVERY. This section is not applicable to this specification.

6.0 NOTES.

(The following information is provided solely for guidance in using this specification. It has no contractual significance).

6.1 Intended Use. This document will apply specifically to the acquisition of computer programs where the acquisition involves either software alone, or software as a portion of a system or subsystem.

6.2 Ordering Data. The procuring activity should consider specifying the following:

6.2.1 Procurement Requirements.

a. Title, number, and date of this specification.

b. Configuration Management. Consideration should be given to citation of MIL-STD-483, Configuration Management Practices For Systems, Equipment, Munitions, and Computer Programs.

c. Software Documentation Standards. Consideration should be given to citing MIL-STD-490, Specification Practices, as amended by Appendix VI, MIL-STD-483 as the standards for software documentation.

d. Technical Reviews and Audits. Consideration should be given to using MIL-STD-1521 as a guide in specifying the number, content, and scheduling of technical reviews and audits.

e. Software QA Program Plan. Consideration should be given to requiring the contractor to deliver a Software QA Program Plan in response to the invitation for bid, or request for proposal, or request for quotation and as a Contract Data Requirements List item (see 6.2.2). The plan should define the methods and procedures which the contractor proposes to use in fulfilling the requirements of this specification.

6.2.2 Contract Data Requirements.

a. The format, type of copy, number of copies, degree of detail required, delivery schedules and purpose of submission will be specified on a DD Form 1423, Contract Data Requirements List, included in the contract for all plans, documentation, and reports which are required to be delivered to the Government. A DD Form 1423 is not required when the Government will review contractor file copies.

Custodians:

Army-AD

Review Activity:

Army-EL
AT
AV
MI
TE

Preparing Activity:

Army-AD

Project No. IPSC-A045

☆U.S. GOVERNMENT PRINTING OFFICE: 1974—713-153/5030

APPENDIX B

SOFTWARE ESTIMATING GUIDELINES

Step 1 -- Determining the Software Type

The estimating task should begin with analysis of the problem to be solved. Generally, the type of software to be developed will be some combination of the following:

-- Mathematical Operations
-- Report Generation
-- Logic Operations
-- Signal Processing, or Data Reduction
-- Real Time, or Executive (also, Avionics Interfacing)

In this step, one should estimate how much of the total software will be of each type, and then apply a reasonableness test to whether this distribution is valid. For example, it would be unusual for more than 10-15% of the software to be in the real time category, or more than half to be in the report generation category.

Step 2 -- Estimating the End Product Size

Next, one should estimate for each of the categories determined in Step 1, how much new code must be written. Unless the project has unusually severe requirements for commentary within the code, this set of estimates can be simply in terms of executable statements.

While not executable in the usual sense, statements which define storage areas (Fortran COMMON, for example) should be included in these estimates. *Ordinarily, these statements will be a relatively small portion* (10% or less) of the total. Care should be taken, however, to recognize situations in which storage definition is a significant portion of the total task -- for example, when the amount of storage available is quite limited and special techniques must be used to fit the data into the space allocated. On the other hand, it should be recognized that storage defining is usually done only once, and then replicated in all of the routines that use the storage.

The estimated number of statements to be coded should include only deliverable code; that number may be significantly smaller than the total number of statements produced, due to the necessity of creating various development aids (test drivers, test data bases, translators, simulators, etc.) to support a project. The adjustment factors of Table B-1 take into account the creation of such aids. *(If a development aid is to be delivered to the customer, it must be considered as a deliverable item in the estimating process, because of the need for testing and documentation of the tool itself).*

The results of this step should be compared with the distribution prepared in Step 1, analyzed for reasonableness, and adjusted if necessary before proceeding to Step 3.

Step 3 -- Estimating the Labor Requirements

The preceding steps will have produced a breakout of the development task in terms of new statements that must be coded in each of several categories or software type. In this step, the following multipliers can be applied to estimate the amount of labor required to produce this code.

--	Mathematical	6 man-months/1000 statements
--	Report	8 man-months/1000 statements
--	Logic	12 man-months/1000 statements
--	Signal	20 man-months/1000 statements
--	Real Time	40 man-months/1000 statements

These factors assume that a "statement" is one fully checked out, tested, and documented statement coded in the selected language. The choice of language can have a significant effect on the development cost, but ordinarily affects only portions of the total task.

Step 4 -- Estimating and Expenditure Distribution

Typically, the final costs of a software development activity will tend to be distributed about as follows:

	Task	% of Total Cost
--	Requirements Definition	5
--	Design and Specification	25
--	Code Preparation	10
--	Code Checkout	25
--	Integration and Test	25
--	System Test	10

The distribution shown is a "raw distribution"; that is, it does not take into account such factors as re-implementation, existence of sophisticated debug tools, etc. These factors are accounted for in Table B-1, and are applied in Step 5. Note that the distribution to Requirements Definition and Design and Specification tasks includes documentation; i.e. the user documentation and detailed design specifications are the product of these two tasks.

Further, while the percentages shown here will, in general, be indicative of the manpower allocated on a project, they will not necessarily represent the actual flow time or calendar time scheduled for each task. The adjusted estimates of Step 5 will more closely approach a flow time distribution and could, therefore, be used as a basis for schedule preparation. Documentation is a very apt example of this problem; while the actual preparation of the technical content of a document is correctly represented in the percentages above, typing support and the flow time and effort involved in producing a finished, printed manual is not and should be added to the estimates produced by these guidelines.

In this step, the labor estimates developed in Step 3 should be broken down by task -- for each of the software types involved in the problem. The resulting n by 6 matrix (where n is the number of different software types and 6 is the number of activities from Step 4) of individual task estimates will be further adjusted in the next step for the particulars of the project.

Step 5 -- Adjusting the Labor Estimates

Table B-1 shows multipliers that should be applied to individual estimates to account for various task characteristics. In using this table, it should be noted that the multipliers are task-specific; for example, use of a higher-order language does not affect Requirements Definition or System Test tasks. Other multipliers are cumulative; for example, use of both a higher-order language and macro capability results in a coding cost which is only 27% of the cost using assembly language without macros.

On the subject of macros, care should be used to avoid including in the estimates the effort required to develop any special algorithms (for instance, to satisfy unusual sizing or timing requirements). Such efforts should be estimated separately and their costs added to the basic project costs estimated using these guidelines. Use of the algorithms can be considered equivalent to using macros, however.

Step 6 -- Estimating Computer Time

After the individual task estimates have been adjusted per Step 5, the revised estimates can be summed to arrive at a total labor cost for the project. The final step is to estimate the machine time that will be used during the development activity.

The most widely accepted rule of thumb is that approximately three hours of stand-alone computer time will be spent per man-month. Since stand-alone time is rarely used except in special cases (mini-computer applications, security requirements), the values from Table B-2 can be used instead.

TABLE B-1: LABOR ESTIMATE ADJUSTMENT FACTORS

			REQUIREMENTS DEFINITION	DESIGN AND SPECIFICATION	CODE PREPARATION	CODE CHECKOUT	INTEGRATION AND TEST	SYSTEM TEST
1.	Re-implementation of existing software		0.2	0.2	0.8		0.8	
2.	Follow-on contract with current customer		0.7					0.9
3.	Number of programmers: (interpolate between values if needed)	1 - 2	0.2	0.5	0.8		0.2	
		6 - 10	1.0	1.0	1.0		1.0	1.0
		more than 20	6.0	3.3	1.2		3.0	3.0
4.	Higher-order language (seasoned compiler)			0.3	0.3	0.2		
5.	Macro-language -- in coding			0.9	0.9	0.9		
	-- forms for document			0.8			0.8	
6.	On-line code/data entry				0.9	0.9		
7.	On-line debugging					0.6		
8.	Poor (or no) debug tools except dumps					1.4	1.4	
9.	Programming experience with engineering/technical discipline of application:	Entry-level	2.0	3.0	1.5			1.5
		Moderate	1.0	1.0	1.0			1.0
		High	0.6	0.5	0.8			0.7

Note: In matrix positions having no entry, the assumed multiplier is 1.0.

—— There are formal procedures for identifying, processing, and tracking problem reports

TABLE B-2: MACHINE TIME ESTIMATING FACTORS

Common Math/Logic	80 CRUs/Man-Month
Executive, Real Time Control	130 CRUs/Man-Month
Critical Functions	200 CRUs/Man-Month

CRU: Computer Resource Unit

APPENDIX C

PROJECT SURVEY QUESTIONNAIRE

PROJECT SURVEY QUESTIONNAIRE

I. Background Information (contractual)

1. Which of the Modern Program Practices given below are used in your software development project? (Check those that apply.)

___ Program Manager Authority--both technical and administrative responsibility for project.

___ Reviews--formal milestone reviews with customer participation at the end of each phase.

___ Unit Development Folders--capture of working materials for each identified item to facilitate end item development, testing, documentation.

___ Design Discipline and Verification--top down design, formal design representation, completion of design, and deliberate verification of design prior to code.

___ Program Modularity--definitions/restrictions on data interfaces between modules, adherence to parent/child relationships between modules.

___ Naming Conventions--structured names for modules and/or data items.

___ Structured Forms--use of Dijkstra forms as supportable in your programming language.

___ Code Verification--deliberate peer reviews of code for each module.

___ Support Libraries and Facilities--use of automated or proceduralized design, coding, and configuration management aids.

___ Phased Testing--defined and formalized unit, functional, and acceptance testing.

___ Configuration Management/Change Control--creation and control of baselines (requirements, design, implementation) and procedures for problem reporting and resolution.

2. For each practice checked, supply the date of adoption of the practice and the associated software development phase and/or milestone at which it was adopted.

3. For each practice checked, provide a brief rationale which tells why the practice was adopted for your software development project. For each practice checked, indicate the degree of success expected or achieved in the project by its adoption.

4. List the computing hardware (including the host and target machines) used by your project and their relationship.

5. List the operating system(s) and compilers/assemblers and utility software used by your project.

6. Indicate the method(s) of access to the hardware and the software. (Check those which apply.)

   ____ Batch

   ____ Remote job entry

   ____ Time sharing

   ____ Stand-alone

7. Describe the availability of hardware/software: the time schedule for computer accessibility and the existence of any restricted or experimental software/hardware used by the project.

8. Indicate the number and type of personnel for your project. (Enter number which applies.)

   ____ Full time, report to Program Manager

   ____ Full time, report outside Program Manager's organization

   ____ Full time, outside contract

   ____ Part time, report to Program Manager

   ____ Part time, report outside Program Manager's organization (e.g., consultants)

   ____ Part time, outside contractor

9. Provide personnel resumes which detail experience and training, both prior to and also during this project development.

10. Indicate the percentage of personnel turnover expected/experienced on your project. Was this turnover planned for? Did it have a detrimental effect on project performance?

11. Characterize your project in terms of magnitude, complexity and software type (e.g., real time, utility, application) performed by the resultant software. When did your project begin?

12. In what phase of the software development is your project currently? (Check only one.)

___ Requirements Definition

___ Design

___ Coding

___ Checkout and Unit Testing

___ Integration and Testing

___ System Testing and Delivery

___ In Production (Operation and Maintenance)

II. Software Estimating Guidelines

1. Estimate how much of the total deliverable software is of each of the following types:

Mathematical Operations ___

Report Generation ___

Logic Operations ___

Signal Processing/Data Reduction ___

Real Time/Executive/Avionics Interfaces ___

How many independent programs does this represent?

2. Estimate the total number of deliverable source statements, excluding commentary. (If this is an enhancement or reimplementation, do not include statements which do not have to be recoded.)

3. Is this development a reimplementation of an existing software design? Is this development a conversion of an existing software system?

4. Is this a follow-on contract with your current customer (e.g., major enhancement)?

5. How many designers/programmers support this development?

___ 1-2 ___ 6-10

___ More than 20

___ 3-5 ___ 11-20

6. Is a higher order language being used?

___ not at all ___ exclusively ___ partially

7. Does the source language used provide a macro capability?

___ not at all ___ yes

8. Do you have documentation forms to aid in expression of designs, tests, data structures, etc.?

___ not at all ___ yes

9. Does the computer system used allow for on-line programming activities?

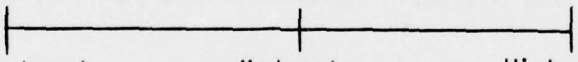
___ not at all ___ yes, code/data entry

___ yes, debugging

10. Does the computer system provide debugging tools?

___ dumps only ___ *other (specify)*

11. What is the designer/programmer experience with the engineering or technical discipline of application? (Insert ∇.)



Entry level Moderate High

12. What is the actual number of man-months (or man-hours) expended (by phase) on this software development to date? What is the actual dollar cost of this labor expense (by phase) to date?

13. What is the actual amount of CRU's expended (by phase) by this software development to date? What is the actual dollar cost of this machine expense (by phase) to date?

14. What is the milestone schedule? What percent of the total flow time is scheduled for the following phases?

Requirements definition ___

Design ___

Code ____

Checkout ____

Integration and functional test ____

System testing and acceptance ____

15. What percent of the total (estimated) manpower required is planned to be (has been) used in each phase?

III. Indicators of Modern Programming Practice

1. What is the responsibility of the Program Manager?

____ Technical (product quality, reliability)

____ Make task assignments

____ Administrative (budgetary)

____ Evaluate performance of personnel

2. Are formal task assignments provided by the Program Manager to project personnel?

3. Are formal phase reviews scheduled and conducted?

____ Pertinent items identified and available at each review

____ Review objectives specified and understood by participants

____ Results of review followed up

____ Project and customer representatives participate in each review

____ The objectives and procedures for this activity were stated in advance

____ This activity was performed according to these objectives and procedures

4. Is project documentation pertinent?

____ Document schedule exists for review and completion

____ Purpose of each document stated and justified

___ Documents are scheduled so that the project can use each completed document as source material for next phase's activities

___ Each document is reviewed by its intended audience

___ Appropriate user content and language are determined for and contained in each document

___ Project control of documentation so that it "tracks" with requirements/design/implementation changes

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

5. Are Unit Development Folders (UDF) used by the project?

___ "Working papers" for each item are captured as they are created

___ Project procedures establish form and content of the UDF's

___ Their contents are utilized for developing other items, completing other tests

___ Project controls the access/use of UDF's

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

6. Does the project employ design discipline?

___ Top down design approach specified

___ Formal design analysis and representation techniques used:

___ Static design representation (design trees)

___ Dynamic design representation (transition diagrams)

___ Other (explain)

___ Design refinement methods are used to adapt the abstract design model to the computing environment

___ Design completeness criteria are developed and specified. What are they?

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

7. Is the design verified?

___ Peer reviews (structured walkthroughs) of design are conducted

___ The objectives and method of conducting peer reviews are stated in advance

___ Design reachability and connectivity analyses procedurized and used

___ Design modularity analyses procedurized and used

___ Mechanical evaluation method used

___ Project control (compliance methods and verification) over the process of design verification exists

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

8. Is the completed design utilized to discipline the code construction process?

___ Standard definition for design and program documentation is established

___ Code construction plan is prepared and used

___ Formal design review involving customer is held prior to start of coding

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

9. Is program modularity practiced?

___ Content and format of program specifications are established

___ Modularity criteria specified and adhered to

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

10. Are structured forms used in the code?

___ Block structures used

___ Permissible logic statements defined, used, and controlled

___ Project compliance and verification procedures control the *use of structured forms*

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

11. Are formal coding conventions being followed?

___ Syntactical forms defined that are (dis)allowed for each programming language

___ Procedures for accessing external data and handling error conditions

___ Naming conventions for units of code and data variables

___ Project control procedures for code organization and comments

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

12. Is the code verifiable?

___ The required testing and examination for each unit of code are documented

___ Peer reviews of the code are procedurized and conducted

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

13. Are design aids to support the software development used?

___ Formal design language used in completing the design

___ Manual aids used (specify)

___ Automated aids used (specify)

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

14. Are code construction aids to support the software development used?

___ Structured programming practices used to complete the code

___ Manual aids used (specify)

___ Automated aids used (e.g., precompiler) (specify)

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

15. Are configuration management aids to support the software development used?

___ Baselined end items are identified and controlled. Indicate at what milestone/phase baseline control was established.

___ Manual aids used (specify)

___ Automated aids used (specify)

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

16. Does the project perform unit/integration software testing?

____ Unit/integration testing is defined in a formal test plan

____ There are expected results and pass/fail criteria defined

____ The software correctly implements the design when this testing is complete

____ There is a problem/error reporting system

____ The project uses compliance and verification methods to control this testing phase

____ The objectives and procedures for this activity were stated in advance

____ This activity was performed according to these objectives and procedures

17. Does the project perform functional software testing?

____ An internal review of items is held to judge their quality and completeness prior to functional testing

____ A handover of items into a controlled configuration is made formally by the developers

____ An independent agency performs functional testing

____ The project produces and executes formal test plans and procedures for functional testing

____ There are expected results and pass/fail criteria defined

____ A realistic dress rehearsal of the acceptance test is performed as the final functional test

____ The software correctly satisfies the requirements when this testing is complete

____ There is a problem/error reporting system

____ The project uses compliance and verification methods to control this testing phase

____ The objectives and procedures for this activity were stated in advance

____ This activity was performed according to these objectives and procedures

18. Does the project perform acceptance testing?

___ Acceptance test requirements are part of the project's requirements baseline

___ Formal acceptance test plan and procedures are developed and user concurrence is obtained

___ There are procedures which allow review of the quality and completeness of the deliverables

___ Project and customer are asked if ready to begin acceptance testing

___ There is a formal deliverables baseline and a committed schedule for acceptance testing as a result of a formal review

___ There is an acceptance test report attesting to the satisfactory conclusion or conditional acceptance

___ The report is signed by the customer

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

19. Does the project use controlled end item baselines?

___ Project end items by phase are identifiable based on written procedural mechanisms

___ Controlled, review-established requirements, design and implementation baselines exist with written procedures

___ There are mechanisms for updating and distributing established baselines

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

20. Does the project utilize a problem reporting system?

___ There are formal procedures for reporting problems, errors, desired improvements

___ There are formal procedures for identifying, processing, and tracking problem reports

___ There are formal procedures for obtaining or distributing problem report status information

___ A file of completed or in progress problem reports exists

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

21. Are baseline change control boards used by the project?

___ The board represents project management and customers

___ They have discretionary and budgetary authority to control changes to the baseline

___ They assess proposed changes, and resolve reported problems by assigning action items

___ They prioritize and control the changes to be implemented

___ They authorize baseline updates and distribution

___ The objectives and procedures for this activity were stated in advance

___ This activity was performed according to these objectives and procedures

APPENDIX D

DERIVATION OF FORECASTED COSTS FOR EACH PROJECT

APPENDIX D

FORECASTED COSTS: PROJECT A

Step 1 Determining Software Type

Mathematical Operations	5%
Report Generation	5%
Logic Operations	30%
Executive	60%

Step 2 Estimating the End Product Size

	HOL	+	Assembly	
37,600 =	28,800		8,800	Source Statements
1,880	Math Ops			
1,880	Report Gen			
11,280	Logic Ops			
22,560	Executive			

Step 3 Estimating the Labor Requirements

(MM)		
11.28	Math Ops	6MM/1000 statements
15.04	Report Gen	8MM/1000 statements
135.36	Logic Ops	12MM/1000 statements
902.40	Executive	40MM/1000 statements
1064.08 MM		

Step 4 Estimating Expenditure Distribution

(MM)				
Math Ops	Report Gen	Logic Ops	Exec	Raw Distribution
0.564	0.752	6.768	45.12	Req Def (5%)
2.820	3.760	33.840	225.60	Design (25%)
1.128	1.504	13.536	90.24	Code (10%)
2.820	3.760	33.840	225.60	Checkout (25%)
2.820	3.760	33.840	225.60	Integ Test (25%)
1.128	1.504	13.536	90.24	Sys Test (10%)

Step 5 Adjusting the Labor Estimates

Adjustment Factors Applying 3(3-5),4,5,6,7

Math Ops	Report Gen	(MM) Logic Ops	Exec	Total	Activity	Multiplier
0.3384	0.4512	4.0608	27.0720	31.9224	Req Def	.6
0.4569	0.6092	5.4821	23.9286	67.0240	Design	.54
0.2467	0.3290	2.9604	12.9215	36.1931	Code	.729
0.2741	0.3655	3.2893	24.6124	50.4697	Checkout	.486
1.3536	1.8048	16.2432	108.2880	127.6896	Integ Test	.48
1.1280	1.5040	13.5360	90.2400	106.4080	Sys Test	1.0
3.7977	5.0637	45.5718	365.2736	419.7068		

$\Sigma =$ 419.7068 MM Total

FORECASTED COSTS: PROJECT B

Step 1 Determining Software Type

Report Generation 65%
Logic Operations 35%

Step 2 Estimating the End Product Size

240,000 (800 modules)x(300 statements per module)
Source Statements All HOL
156,000 Report Generation
84,000 Logic Operations

Step 3 Estimating the Labor Requirements

(MM)		
1248	Report Gen	8MM/1000 statements
1008	Logic Ops	12MM/1000 statements
2256 MM		

Step 4 Estimating Expenditure Distribution

(MM) Report Gen	Logic Ops	Raw Distribution
62.4	50.4	Req Def (5%)
312.0	252.0	Design (25%)
124.8	100.8	Code (10%)
312.0	252.0	Checkout (25%)
312.0	252.0	Integ Test (25%)
124.8	100.8	Sys Test (10%)

Step 5 Adjusting the Labor Estimates

Adjustment Factors Applying 1,2,3(>20),4,6,7

Report Gen	(MM) Logic Ops	Total	Activity	Multiplier
52.416	42.336	94.7520	Req Def	.84
61.776	49.896	111.6720	Design	.198
32.3482	26.1274	58.4756	Code	.2592
33.696	27.216	60.9120	Checkout	.108
748.8	604.8	1353.6000	Integ Test	2.4
336.96	272.16	609.1200	Sys Test	2.7
1265.9962	1022.5354	2288.5316		

$\Sigma =$ 2288.5316 MM Total

FORECASTED COSTS: PROJECT C

Step 1 Determining Software Type

Mathematical Operations	2%
Report Generation	80%
Logic Operations	10%
Data Reduction	8%

Step 2 Estimating the End Product Size

20,000	Source Statements All HOL
400	Math Operations
16,000	Report Generation
2,000	Logic Operations
1,600	Data Reduction

Step 3 Estimating the Labor Requirements

(MM)		
2.4	Math Ops	6MM/1000 statements
128.0	Report Gen	8MM/1000 statements
24.0	Logic Ops	12MM/1000 statements
32.0	Data Reduct	20MM/1000 statements
186.4 MM		

Step 4 Estimating Expenditure Distribution

(MM) Math Ops	Report Gen	Logic Opst	Data Reduct	Raw Distribution
0.12	6.4	1.2	1.6	Req Def (5%)
0.60	32.0	6.0	8.0	Design (25%)
0.24	12.8	2.4	3.2	Code (10%)
0.60	32.0	6.0	8.0	Checkout (25%)
0.60	32.0	6.0	8.0	Integ Test (25%)
0.24	12.8	2.4	3.2	Sys Test (10%)

Step 5 Adjusting the Labor Estimates

Adjustments Factors Applying 2,3(3-5),4,5,6,7,9

(MM)

Math Ops	Report Gen	Logic Ops	Data Reduct	Total	Activity	Multiplier
0.0403	2.1504	0.4032	0.5376	3.1315	Req Def	.3360
0.0729	3.8880	0.7290	0.9720	5.6619	Design	.1215
0.0472	2.5190	0.4723	0.6298	3.6683	Code	.1968
0.0583	3.1104	0.5832	0.7776	4.5295	Checkout	.0972
0.2880	15.3600	2.8800	3.8400	22.3680	Integ Test	.4800
0.1561	8.3238	1.5607	2.0810	12.1216	Sys Test	.6503
0.6628	35.3516	6.6284	8.8380	51.4808		

$\Sigma =$ 51.4808 MM Total

FORECASTED COSTS: PROJECT D

Step 1 Determining Software Type

Report Generation	5%
Logic Operations	90%
Data Reduction	5%

Step 2 Estimating the End Product Size

198,000 = (330 modules) x (600 statements per module)

Source Statements All HOL	
9,900	Report Generation
178,200	Logic Operations
9,900	Data Reduction

Step 3 Estimating the Labor Requirements

(MM)		
79.2	Report Gen	8MM/1000 statements
2138.4	Logic Ops	12MM/1000 statements
198.0	Data Reduct	20MM/1000 statements
2415.6 MM		

Step 4 Estimating Expenditure Distribution

(MM) Report Gen	Logic Ops	Data Reduct	Raw Distribution
3.96	106.92	9.9	Req Def (5%)
19.80	534.60	49.5	Design (25%)
7.92	213.84	19.8	Code (10%)
19.80	534.60	49.5	Checkout (25%)
19.80	534.60	49.5	Integ Test (25%)
7.92	213.84	19.8	Sys Test (10%)

Step 5 Adjusting the Labor Estimates

Adjustment Factors Applying 2,3(>20),4,5

Report Gen	(MM) Logic Ops	Data Reduct	Total	Activity	Multiplier
16.632	449.064	41.580	507.276	Req Def	4.2
14.11344	381.06288	35.2836	430.45992	Design	.7128
2.56608	69.28416	6.4152	78.26544	Code	.324
3.564	96.228	8.91	108.702	Checkout	.18
47.52	1283.04	118.8	1449.36	Integ Test	2.4
23.76	641.52	59.4	724.68	Sys Test	3.0
108.15552	2920.19904	270.3888	3298.74336		

Σ = 3298.74336 MM Total

FORECASTED COSTS: PROJECT E

Step 1 Determining Software Type

Report Generation	22%
Logic Operations	71%
Executive	7%

Step 2 Estimating the End Product Size

3,390	Source Statements All HOL
625	Report Generation New
2,055	Logic Operations New
210	Executive New
2,890	Total changed source statements

Step 3 Estimating Labor Requirements

(MM)		
5.0	Report Gen	8MM/1000 statements
24.66	Logic Ops	12MM/1000 statements
8.4	Exec	40MM/1000 statements
38.06		

Step 4 Estimating Expenditure Distribution

(MM) Report Gen	Logic Ops	Exec	Raw Distribution
.25	1.2330	.42	Req Def (5%)
1.25	6.1650	2.10	Design (25%
.50	2.4660	.84	Code (10%)
1.25	6.1650	2.10	Checkout (25%)
1.25	6.1650	2.10	Integ Test (25%)
.50	2.4660	.84	Sys Test (10%)

Step 5 Adjusting the Labor Estimates

Adjustment Factors Applying 1,2,3(1),4

Report Gen	(MM) Logic Ops	Exec	Total	Activity	Multiplier
.0070	.0345	.0118	0.0533	Req Def	.028
.0375	.1850	.0630	0.2855	Design	.030
.0960	.4735	.1613	0.7308	Code	.1920
.2500	1.2330	.4200	1.9030	Checkout	.2
.2000	.9864	.3360	1.5224	Integ Test	.16
.4500	2.2194	.7560	3.4254	Sys Test	.9
1.0405	5.1318	1.7481	7.9204		

$\Sigma =$ 7.9204 MM Total

APPENDIX E

SOFTWARE PROJECTS SELECTED FOR ANALYSIS

APPENDIX E

SOFTWARE PROJECTS SELECTED FOR ANALYSIS

Five in-house developmental software projects were selected to provide the software production data we collected. These projects vary in size, complexity, and the extent to which both traditional and modern programming practices were exploited.

The next two sections provide functional descriptions of each project, including objectives, special features, and information flow. Following this, descriptions of each project's computing environment, personnel background, and the parameters used to forecast development costs are presented.

In order to provide the greatest amount of project specific information without compromising the identity of any one project, there is no relation between the names and the order of the functional description of these projects, and the project labels A, B, C, D, E used in the following descriptions of project characteristics.

PROJECT FUNCTIONAL DESCRIPTIONS

CASA (Configuration Accountability System Aerospace) is the first of a four-phase development to redesign and upgrade the computer support for The Boeing Aerospace Company (BAC) engineering and operations divisions. BAC's computer processing support, which originated in the 1950's and early 1960's, was studied in 1973. The study concluded that significant improvements could be achieved by eliminating fragmentation in computers and systems, exploiting recent advances in computer technology, and redesigning software instead of converting it to newer hardware. Consequently, BAC/BCS decided to develop systems that would better support BAC objectives and take advantage of improved computerized support.

CASA is primarily designed to support engineering in the establishment and maintenance of design configuration information. In addition, other organizations such as manufacturing, material, quality control, and finance are supported by the system. The system serves as a baseline for all manufacturing and procurement activities and creates program unique parts lists that form the basis for configuration control, maintenance of engineering release records, and easier accessibility to current engineering data. Figure E-1 shows the integrated data base, products and organizational interfaces for CASA.

CASA is comprised of four major subsystems: Automated Release Records, Automated Parts List. Parts Usage, and Configuration Identification. In supporting engineering design, configuration control and release unit activities, these subsystems reduce the average flow time between initial release of a requirement and hardware delivery to the customer from 48 to 34 calendar days.

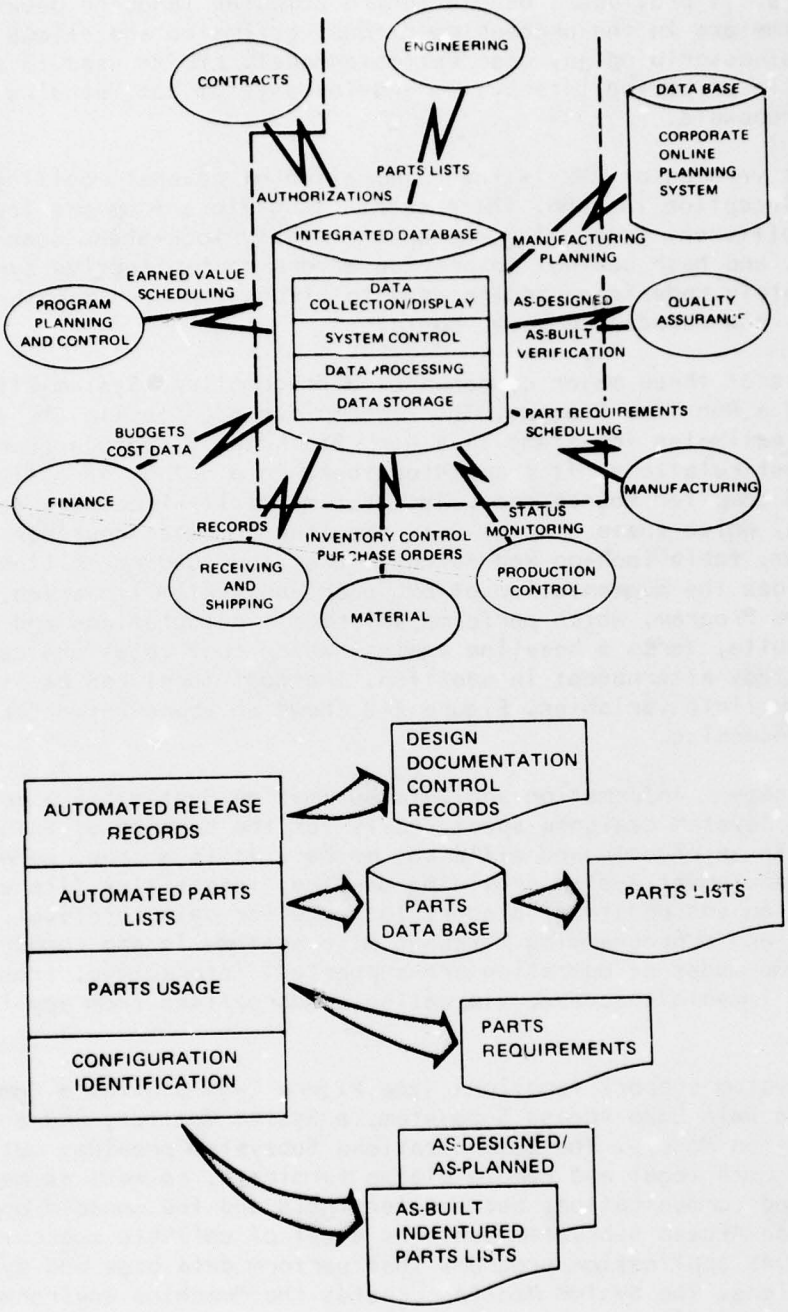


FIGURE E-1. CONFIGURATION ACCOUNTABILITY SYSTEM AEROSPACE (CASA)

EML (Estimator Modeling Language) is a software system used to estimate system costs. It provides a user-oriented computer language developed for use by estimators in the processing of cost estimates and allows a convenient means of describing any cost estimate model. EML is used to aid cost estimators in preparing intersystem and intrasystem cost studies, cost trades and cost proposals.

The present version of EML is the culmination of several modifications since its inception in 1966. The present EML differs from previous versions by use of different processing concepts (notably look-ahead scanning, list processing, and hash coding) to provide a more cost-effective system. It has been completely redesigned and recoded to afford improved maintainability, flexibility and reduced computer costs.

EML consists of three major components: a Precompiler, System Utility Processes, and a Run Time Program. The Precompiler accepts (in EML language form) cost estimates for items in a Work Breakdown Structure expressed in terms of cost relations. It translates these into output directives, initialization and compiler source code. The System Utility Processes consist of compilation, which fuses special user supplied compiler routines with the main program, table look-up and calculations, and loading or link editing, which provides the augmentation of EML user and system libraries. The resulting Run Time Program, which performs arithmetic calculations and produces tabular results, forms a baseline against which cost relations can be varied for trade study alternates; in addition, the cost model can be iterated for single or multiple variables. Figure E-2 shows an abbreviated EML system flow and processing.

MIDAS (Management Information and Data Automation System) is a powerful and flexible system designed specifically for the purpose of information processing in an orderly and efficient manner. It is a true, generalized data base management system providing on-line, interactive file creation, data insertion and updating, a query language for data retrieval, report generation, and a programming language with arithmetic and comparison capabilities. Two modes of operation are supported: interactive, from user terminals, and immediate access, via callable subroutines from application programs.

The MIDAS system support functions (see Figure E-3) provide a Communications Subsystem, a Data Base Access Subsystem, a System Monitor, and a Terminal Task Foundation Module. The Communications Subsystem provides multi-terminal support for both local and remote dialup terminals, as well as message switching and communications between terminals and the console operator. The Data Base Access Subsystem contains a set of callable subroutines available for MIDAS application programs that perform data base and system oriented functions. The System Monitor creates the "machine environment" necessary for MIDAS implementation, augmenting the resident disk operating system and including frequently used general utility functions and routines necessary to allocate CPU cycles, memory, and other system resources. The Terminal Task Foundation Module provides the basic interactive user environment and contains seven processors for: System Control, Data Definition,

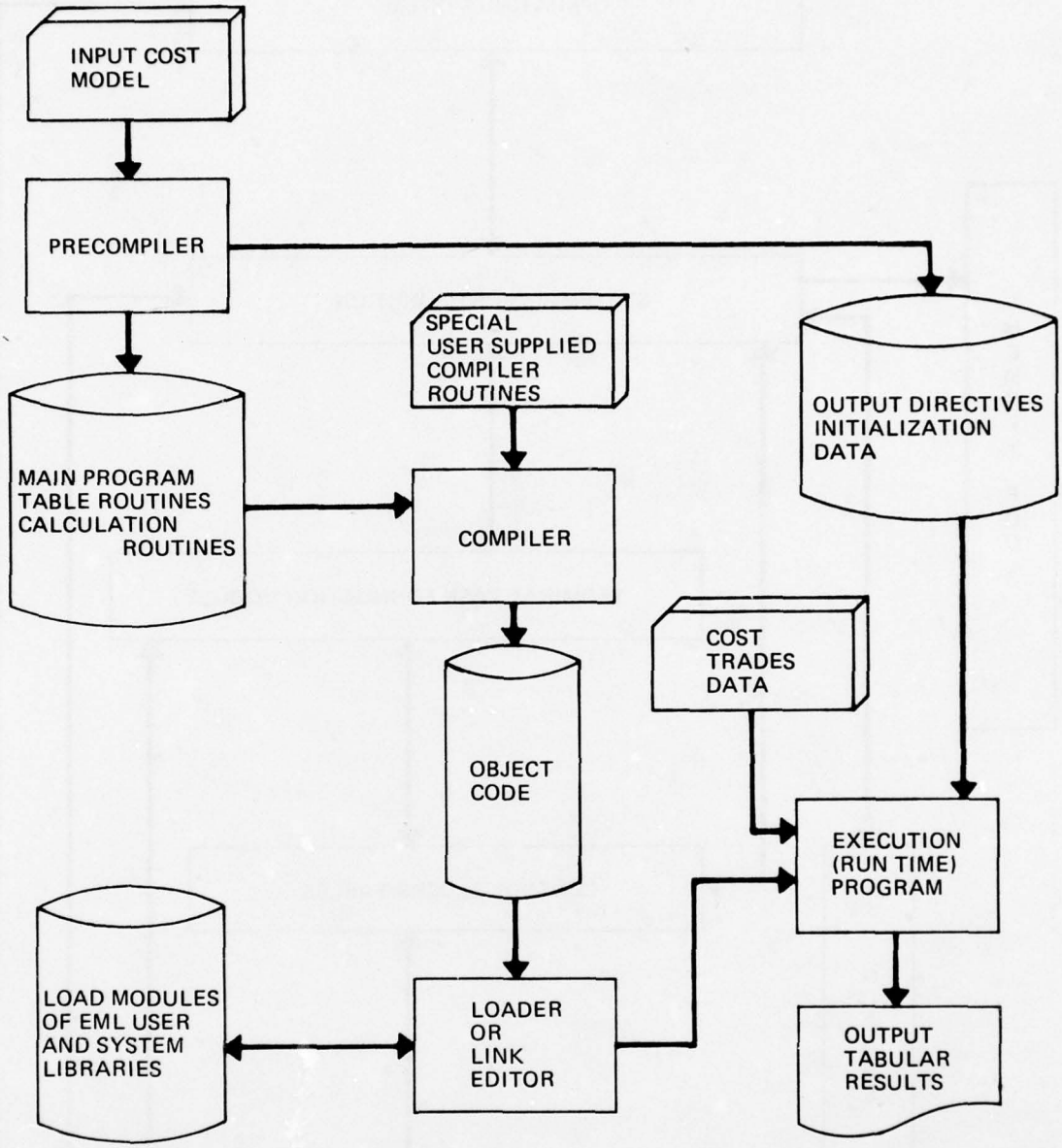


FIGURE E-2. ESTIMATOR MODELING LANGUAGE (EML)

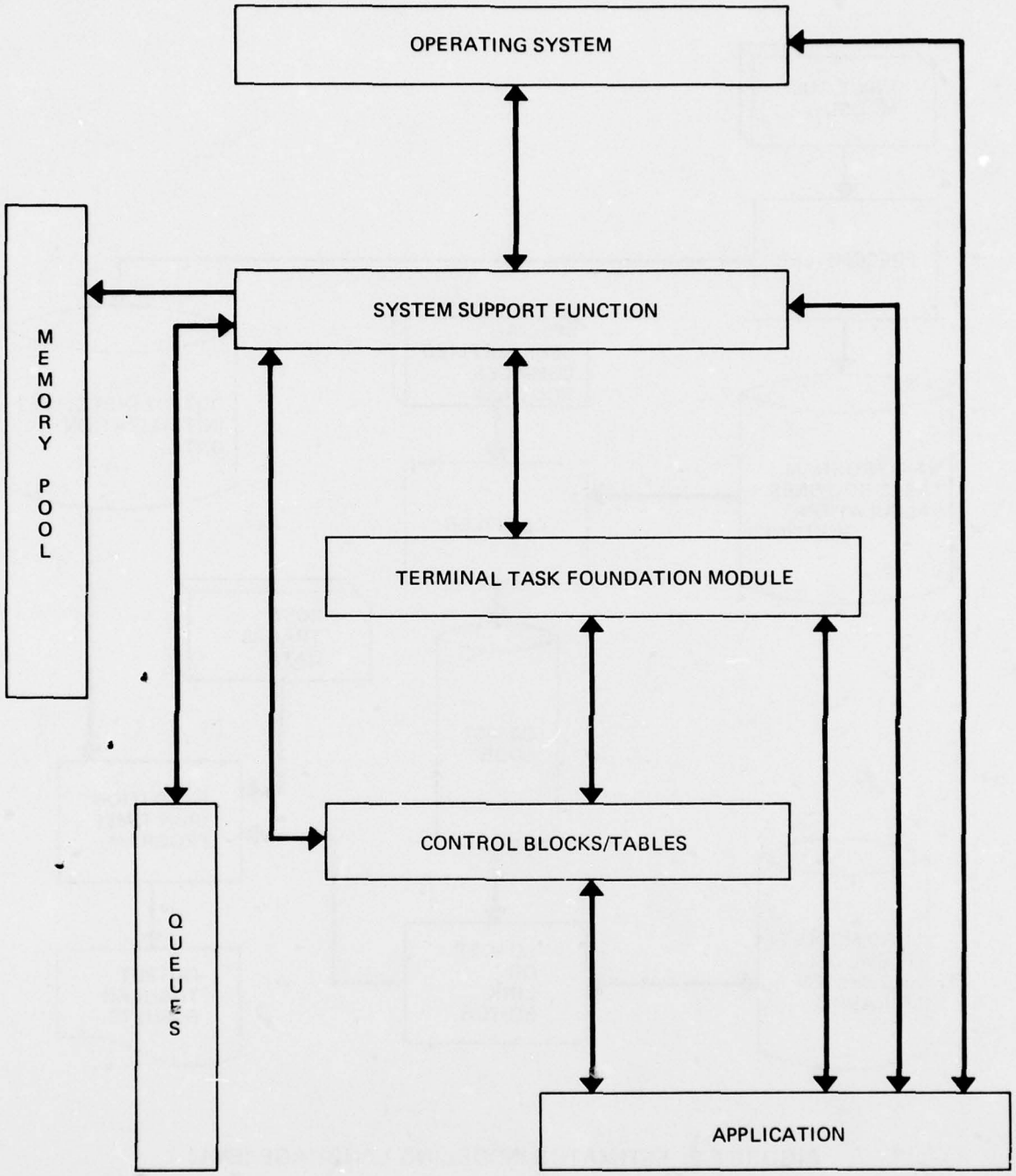


FIGURE E-3. MANAGEMENT INFORMATION AND DATA AUTOMATION SYSTEM (MIDAS)

Data Manipulation, Query Language, Report Generation, MIDAS Program Language, and Remote Job Entry.

Features of MIDAS include an integrated data base, minimal redundancy data sharing, biasing of data to multiple applications, a variety of access procedures, a data base administrator with control functions, a variety of data structures, direct user-computer interface, selectable/automatic report formatting, dynamic file management, and comprehensive selectable data and system security.

SARA III (System Analysis and Resource Accounting) provides a capability for abstracting, organizing, and reporting -- from raw accounting data collected by an operating system -- information describing the characteristics of the workload presented to a computing system, and the performance of the system in processing that workload.

The original SARA software evolved over several years as a tool to aid system analysis -- first, to see what performance indicators could be derived from accounting data, and then as a possible substitute for more direct (and costly) hardware/software measurements. The implementation of SARA as a single, comprehensive performance evaluator resulted in this program becoming the standard tool in BCS for evaluating computer system performance.

Because it "just grew," maintenance of the SARA software gradually became quite costly. In an attempt to overcome this problem, the existing capability was re-implemented -- as SARA II -- using structured forms in the source code. While this resulted in more maintainable software, SARA/SARA II remained rather cumbersome to use.

SARA III development had several objectives: to simplify preparation of necessary control statements, to substantially improve the utility of its reports, and to provide a means whereby users could easily obtain specialized reports. In addition, this development was undertaken to provide a more adaptable tool for a variety of operating systems and their associated job accounting mechanisms. In addition to satisfying these basic objectives, the SARA III development involved the application of top down design techniques.

SARA III functions include Accounting Data Reduction (involving timing, resource allocation and CPU utilization), and Statistical Report Generation, based on a selective control language allowing default parameters. Figure E-4 shows an abbreviated SARA III system flow diagram.

WIRS (Wire Information and Release System) was developed to support Boeing Commercial Airplane Company's Engineering and Manufacturing activities, including: the design and release of wire bundle assemblies to manufacturing, the assembly and test of wire bundle assemblies, and the control of the configuration of wire bundles through the design and manufacturing processes.

WIRS provides visibility and status of design, planning and manufacturing activities, release orders, and serves as the data bank for wire bundle

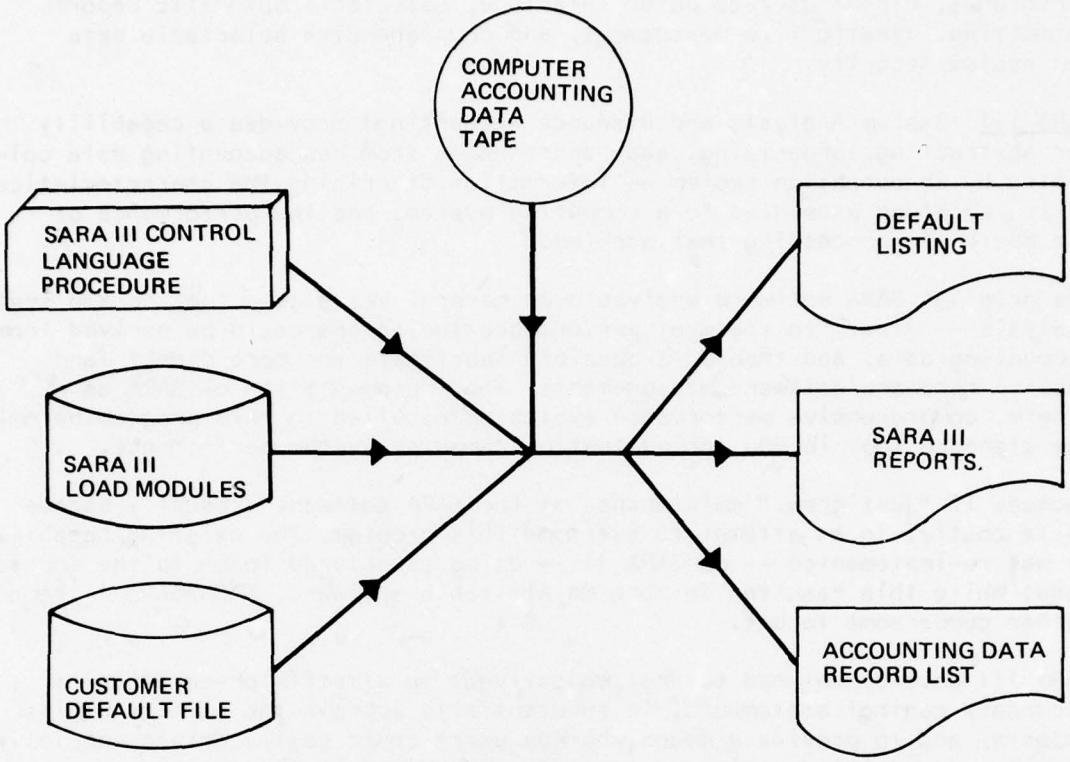


FIGURE E-4. SYSTEMS ANALYSIS AND RESOURCE ACCOUNTING (SARA III)

records. Input to WIRS consists primarily of manual inputs via CRT equipment from Wire Bundle Assemblies (WBA) provided by integration engineers. Output from WIRS consists principally of Wire Bundle Assemblies, control of status reports, shop aid packages and CRT displays for the engineer, planner or fabricator. Figure E-5 shows the relationships within the system.

The system is terminal (data) driven and uses a vendor-supplied data base management capability. The system replaces an existing wire release system and interfaces with an overall on-line planning system. Because of heavy random access requirements, the principal access method is via hierarchical direct access; however, for some reporting processes, sequential access methods are used.

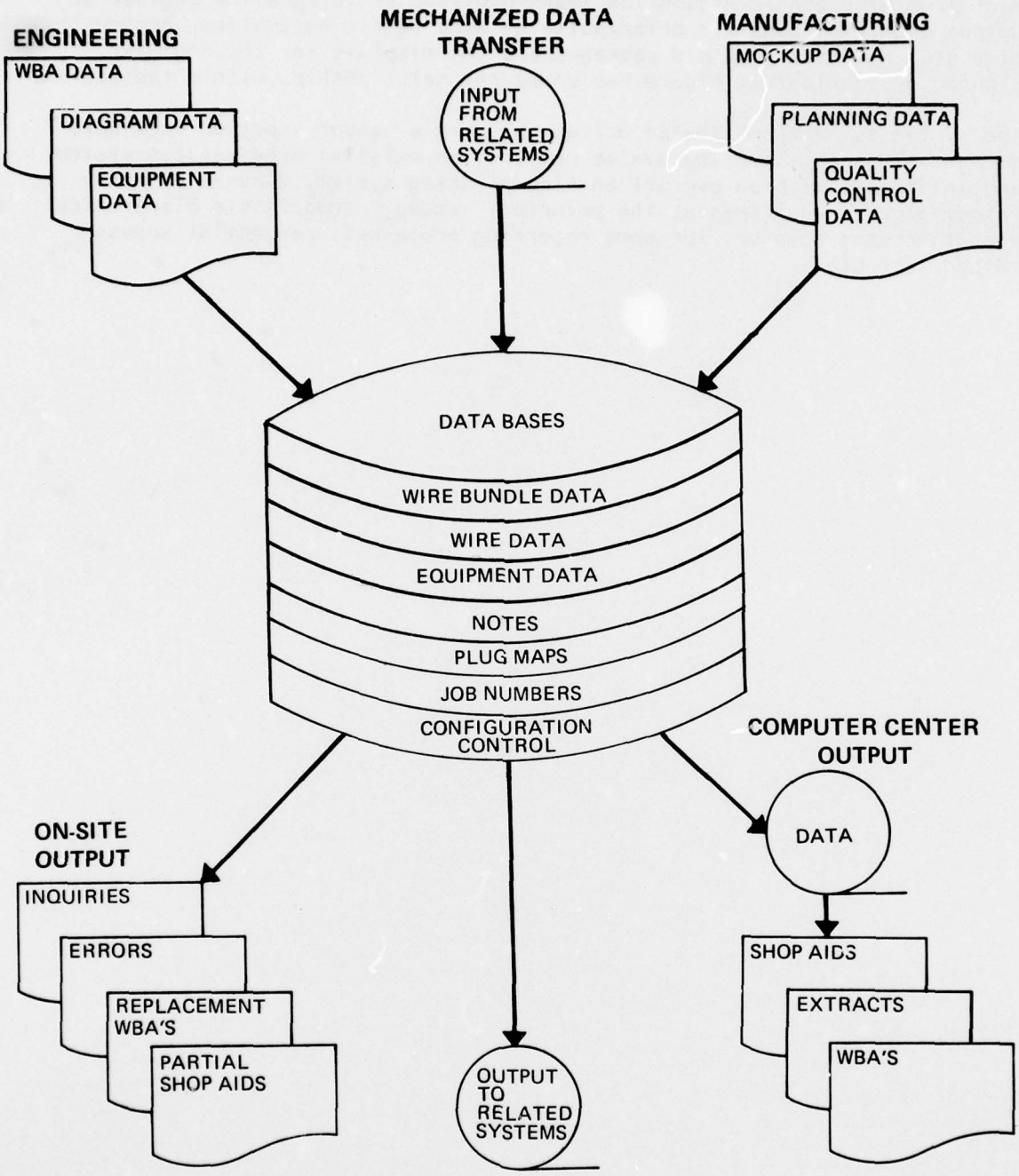


FIGURE E-5. WIRE INFORMATION AND RELEASE SYSTEM (WIRS)

PROJECT CHARACTERISTICS DESCRIPTIONS

COMPUTER CHARACTERISTICS

The computer hardware used on the projects that were studied was predominantly IBM 360 and 370 machines; one project used the NOVA/830 minicomputer. Two projects each used Cobol and Fortran IV languages, while Algol was used in one project. Machine access methods reported included stand-alone, remote batch, remote job entry, and on-line terminal. Figure E-6 details the computer characteristics for each project.

PERSONNEL CHARACTERISTICS

Project A was composed of five members including the Program Manager. One member was an outside contractor and supported the project part-time. In terms of technical experience and training, one member had only minicomputer experience; another had some minicomputer experience (filled out by large-scale computer experience); while the rest had only large-scale computer experience. None of the members brought with them any Algol background. All project personnel received training in top down design and structured programming.

Project B currently consists of 20 members reporting to the Program Manager; personnel size has fluctuated from 2 through 27. This project experienced a rather severe and abrupt personnel turnover immediately after Critical Design Review. The Program Manager stated that their design documentation and configuration management practices cushioned the shock of the turnover and a greater amount of productive work was salvageable.

Training of personnel for Project B included courses in Cobol, IMS, and top down design and structured programming. Less formal training was also provided in the areas of Job Control Language procedures, Data Management Systems, and project specific procedures.

Project C was made up of five members (on the average) including the Program Manager. At its height, seven members were supporting the project. Of these, two reported to the Program Manager, four reported outside the Program Manager's organization, one was an outside contractor, and one was a part-time consultant. In addition, the Program Manager was replaced midway through the project, and the identified customer was changed at about the same time. All of the personnel had large-scale (IBM 360) computer experience, and were familiar with Fortran. The four design team members had been trained in top down techniques prior to assignment to the project.

Project D consists of 37 full-time project members of which 19 report to the Program Manager, 12 report outside the Program Manager's organization, and 6 are outside contractors possessing IMS expertise. All project personnel received training in top down design and structured programming, and in data base management concepts. They were all experienced in Cobol, and 75% of them had prior experience in a large-scale (IBM) environment. The

FIGURE E-6 COMPUTER HARDWARE, SOFTWARE TYPES AND ACCESSIBILITY CHARACTERISTICS BY PROJECT

PROJECT	PRINCIPAL HARDWARE		SUPPORT SOFTWARE	MACHINE ACCESS
	HOST	TARGET		
A	Data General NOVA/830	Data General NOVA/830 Compatible	RDOS Operating System, MAC Macro Assembler, Algol Compiler, Text Editor, Debugger, Precompiler	Time-Sharing; Stand-alone
B	IBM 370/158 (VS 1.0) IBM 370/168 (VS 2.4)	IBM 370/168 or 370/158 (MVS)	Cobol Compiler, BAL Assembler, IMS, TSO, CTS (Time Sharing), BTS (Batch Terminal Simulated, IEBUPDATE, KAPEX Cobol Optimizer	RJE; Time-Sharing
C	IBM 360/65 (OS/MVT)	IBM 360 Compatible	OS/MVT, MAINSTREAM TSO, Fortran IV (Level H) with Precompiler, Programming Support Library Aid	RJE; Time-Sharing
D	IBM 370/158 (VS 1.0) 370/168 (VS 2.4)	IBM 370/158 or 370/168	Cobol Compiler, BAL Assembler, MARK IV utilities, MAINSTREAM TSO, IMS	RJE; Time-Sharing
E	IBM 370/65 370/158 370/145	IBM 360, 370 Compatible	Fortran IV (Level H) Compiler, TSO, CTS (Time Sharing)	Remote Batch (Mailed Decks)

project has thus far experienced a 10% personnel turnover rate, consisting principally of planned and phased entry and departure of outside contractor personnel.

Project E consisted of the Program Manager and one programmer who did not report directly to the Program Manager. The programmer received training in top down design and structured programming before the project began, and had extensive Fortran language and large-scale (IBM) computer experience. In addition, he had supported an earlier version of the software product which was to be re-developed.

APPENDIX F

GLOSSARY

acceptance	Delivery and demonstration of a capability (including software, documentation, and supporting materials) to the customer, in order to confirm that the package is ready for operational usage and that the software developer has satisfied his contractual obligation.
activity	A group of tasks of a similar nature, which will be performed in parallel over a period of time (e.g. coding activity).
aggregate project	A project whose organization is such that all project personnel report directly to a project manager, including staff personnel (scheduling, task supervision, funding supervision, site activation) and functional personnel (design, test planning and conduct, programming, documentation).
algorithm	A procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation.
baseline	A list of the configuration items which provide a designated set of capabilities.
capability, computing	The ability to use a computer to perform functions which support a mission. Generally, a computing capability includes instructions (code) to be processed by the computer, documentation describing how to use the capability, and training for personnel associated with the mission.
CDR	Critical Design Review (See Section 3.1.2).
checkout	The activity of testing code to ensure that it accurately reflects the intent of the design from which the code was produced.
client	An intra-company organization that requires computing services, especially development of software.
code	n, the instructions which direct a computer to perform a computing operation; v, the act of preparing computer instructions.
component (functional component)	A defined constituent of a computing capability, which implements a particular subset of required functions, i.e. a computer program, or document, or data base.

concurrence	Acknowledgement of the receipt of specified items, and the submittal of review comments on those items.
constituent	A product (resulting from a task) which will be incorporated in or used to build an end item.
control listings	The computer listings of input and output resulting from an acceptance test dress rehearsal, indexed to facilitate comparison with the results obtained during formal demonstration.
Critical Design Review	That milestone review which is held immediately prior to the start of software construction, for the purpose of determining that necessary designs and plans have been prepared and are adequate for governing construction activities.
customer	An agency or company that has contracted for development of a computing capability, including software.
Definition	The phase of software development which includes the activities of requirements definition and product specification.
deliverable	An end item which is specified (by means of a Contract Data Requirements List) to be delivered to the customer in fulfillment of a contractual obligation.
demonstration	The activity of showing or proving the operational readiness of a capability (software, documentation, supporting materials) in order to obtain customer acceptance.
design	n, the activity of defining logic and control flow, interfaces, calling sequences, and input and output for software elements (subroutines or components); v, the product (documented design representation) of this activity.
Design	The phase of software development which includes the activities of detailed design, design verification, and planning for testing, installation, operation, and training activities.
design verification	The review of a design to ensure requirements satisfaction, consistency, and completeness.
developer (software developer).	The organization or company supplying software and computing services.

end item	A final result of a development process; specifically, a development product in a form suitable for delivery and operational use (i.e. a deliverable, q.v.)
functional organization	An organization which performs a common business function within a company (e.g. marketing, engineering, manufacturing, finance).
integration	The act of determining that interfacing functional components are mutually compatible and together perform their specified functions correctly.
integrator	The individual assigned to perform integration.
Installation Manual	A document detailing the procedure used to install a capability at a computing facility, and tailor it for local conditions.
Interface Working Group	Two or more programmers (each assigned responsibility for development of functional components) who define the communication protocol and interfaces between components of software.
intermix project	An organizational structure where staff functions (scheduling, task and funds supervision) and some of the primary project-specific functions (engineering, manufacturing) report to the project manager.
kernels	Those constituents of a software component which are considered most technically difficult or critical to the overall function of a component.
lead programmer	The individual assigned the responsibility for identifying major functional components of a computing capability and co-ordinating the assignment of component design and construction tasks to other programmers.
line manager	The person having administrative responsibility for technical personnel.
linkage conventions	The rules of code construction which apply when software invokes a subsidiary constituent (i.e. a subroutine).
locking up	The process of placing software code in a controlled library or file, so that only authorized personnel may make changes.
Maintenance Manual	That document which presents all information regarding design and implementation details of a computing capability which are required to effect repairs or enhancements of that capability (See Section 2.2.2).

AD-A039 852 BOEING COMPUTER SERVICES INC SEATTLE WASH F/G 9/2
BCS SOFTWARE PRODUCTION DATA.(U)
MAR 77 R K BLACK, R KATZ, M D GRAY F30602-76-C-0174
UNCLASSIFIED BCS-40151 RADC-TR-77-116 NL

3 OF 3
AD
A039852

END
DATE
FILMED
6-77

milestone	A significant event of mutual concern to the developer and user/client/customer of a computing capability.
milestone review	A joint meeting between customer and developer for the purpose of reviewing accomplishments to date and plans for subsequent activities. Typical milestone reviews in a software development are Preliminary Design Review (PDR), Critical Design Review (CDR), and Physical Configuration Audit/Functional Configuration Audit (PCA/FCA).
module	A closed subroutine which has the potential of being independently compiled, and may be called by another module in a program.
Operation Manual	That document containing instructions (to computer operations personnel) for performing those tasks necessary to keep a computing capability in operational condition. Specifically, such a document will define necessary data and file manipulations, backup and recovery procedures, and fault diagnosis.
PCA/FCA	Physical Configuration Audit/Functional Configuration Audit (See Section 3.1.2).
PDR	Preliminary Design Review (See Section 3.1.2).
performance	See status/progress/performance.
Physical Configuration Audit/Functional Configuration Audit	That milestone review which is held prior to the start of acceptance testing, to determine that all end items have been completed and verified for functional adequacy.
PM	Program Manager (See Section 3.1.1).
Preliminary Design Review	That milestone review which is held prior to the start of software design, to determine that documented requirements adequately define the computing capability needed, and that the products specified to provide the required capability are necessary and sufficient.
primitives	Those constituents of a design which are directly implementable by a block of compilable source statements with only simple control logic, or by an "off the shelf" software element (e.g. library subroutine, operating system service, utility).
problem	A request for a repair or enhancement to (one or more constituent parts of) a computing capability.

Product Specification	A document in the form of a User Guide which describes the planned use and operation of a capability. Typically, this document will specify planned input and output formats and will identify, in addition to computing software, the other products (documents, training materials, data sets) necessary to complete the capability.
progress	See status/progress/performance.
project organization	An organization of company resources to develop a product or new capability, or to prepare for manufacture of same.
staff project	An organization structure where only staff functions (e.g. scheduling, task supervision, funds supervision) report directly to the project manager.

status/progress/performance

status	The determination of which tasks of a planned activity have been completed.
progress	The comparison of status against schedule.
performance	The comparison of status and expenditure against schedule and budget.

structured walkthrough	A method of design verification, involving the review of one individual's design by a peer, a lead programmer, or a technical customer representative.
subroutine	A software constituent of a functional component, which performs one or more subfunctions necessary during component execution. A subroutine may not have the potential of being separately compiled, and may be called by another subroutine or a module.
test driver	A program created to exercise other software elements for testing purposes.
test objective	The specific requirements which a particular test is intended to demonstrate have been satisfied by the computing capability.
test procedure	A step-by-step set of instructions for setting up, conducting, and determining the outcome of a test.
UDF	Unit Development Folder (See Section 3.2.2).

User Guide — That document which describes how to use a computing capability to perform a required function. Specifically, such a document will describe how to prepare necessary input data, how to cause execution of the capability, and how to interpret the resulting output.

METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm		V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10^{12}	tera	T
1 000 000 000 = 10^{9}	giga	G
1 000 000 = 10^{6}	mega	M
1 000 = 10^{3}	kilo	k
100 = 10^{2}	hecto*	h
10 = 10^{1}	deka*	da
0.1 = 10^{-1}	deci*	d
0.01 = 10^{-2}	centi*	c
0.001 = 10^{-3}	milli	m
0.000 001 = 10^{-6}	micro	μ
0.000 000 001 = 10^{-9}	nano	n
0.000 000 000 001 = 10^{-12}	pico	p
0.000 000 000 000 001 = 10^{-15}	femto	f
0.000 000 000 000 000 001 = 10^{-18}	atto	a

* To be avoided where possible.

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.